

Master Thesis

Visual Comparison of Business Process Flowcharts

Bernhard Häussner

Date of Submission: April 19, 2017
Advisors: Prof. Dr. Alexander Wolff
Fabian Lipp, M. Sc.



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen, Komplexität und wissensbasierte Systeme

Abstract

Increasingly, processes and relationships of many aspects of our lives are described using abstract models. Graphs are the underlying principle behind some of these models. Companies model business processes using event-driven process chains (EPCs). These EPCs can be interpreted as graphs, thus technology known for visualizing graphs can be leveraged for use with EPCs, too. One case of using EPCs is the comparison of different yet similar models. Because EPCs can represent complex behaviors, algorithmic comparison aids are developed to help manage these complexities. These assistant algorithms can often be applied to EPCs and graphs alike based on the relationship between EPCs and graphs. This way, they can transcend into many domains where graphs are already an established tool. Three algorithmic comparison aids are further investigated. One emerged from a tool developed by Andrews et al. [AWW09], which calculates and visualizes a merged graph. The other ones rely on finding a maximum independent set of a permutation graph and were developed to use existing visualizations of models. The drawings are adjusted to bring related parts of the model closely together, at least in one dimension. Lastly, these algorithmic comparison aids are evaluated to verify the theoretical advantages in real-life scenarios. Participants of a user study rated usability and helpfulness of the methods.

Zusammenfassung

Zunehmend werden Abläufe und Zusammenhänge aus allen Lebensbereichen mithilfe von abstrakten Modellen beschrieben. Graphen bilden die mathematische Grundlage für einige dieser Modelle. Geschäftsprozesse in Unternehmen werden als Ereignisgesteuerte Prozesskette (EPK) modelliert. Solche EPKs können in Graphen überführt werden und damit werden bekannte Techniken zur Visualisierung von Graphen auch für EPKs nutzbar. Beim Umgang mit EPKs werden manchmal verschiedene Modelle manuell verglichen. Da die EPKs sehr komplex sein können, lohnt es sich, automatische Assistenzsysteme zu entwickeln, um Komplexitäten dabei handhabbar zu machen. Diese Assistenzsysteme können aufgrund der Ähnlichkeit von EPKs und allgemeinen Graphen auch auf den Vergleich von Graphen allgemein übertragen werden, was eine Anwendung in verschiedenen Domänen auch außerhalb der Modellierung von Geschäftsprozessen erlaubt. Drei dieser Assistenzsysteme werden genauer beschrieben, darunter eine Methodik von Andrews et al. [AWW09], welche auf dem Berechnen und Visualisieren eines zusammen-

gefügten Graphen basiert. Außerdem werden zwei neu entwickelte Methodiken basierend auf maximalen stabile Mengen von Permutationsgraphen vorgestellt, welche bereits vorhandene Visualisierungen der Modelle so aneinander anpassen, dass korrelierende Modellteile in einer Dimension räumlich näher aneinander gebracht werden. Zuletzt wird untersucht, inwiefern solche theoretisch erdachten Assistenzsysteme Nutzern im praktischen Einsatz helfen können, unter anderem mit einer Benutzerstudie, bei der die Teilnehmer die verschiedenen Verfahren hinsichtlich intuitiver Bedienbarkeit und Nützlichkeit bewerten.

Contents

1	Introduction	5
2	Graphs and Business Processes	9
2.1	Graphs	9
2.2	Business processes and event-driven process chains	11
2.3	Finding or specifying similarities	13
2.4	Sugiyama’s layout algorithm	14
3	Placing Similar Vertices on the Same Height	17
3.1	A simple adjustment to make two constraints horizontal	17
3.2	Conflicts between n constraints	19
3.3	Selecting as many constraints as possible	20
3.4	Finding a maximum independent set of a permutation graph	22
3.5	Calculating offset lists for adjusting the layout	27
3.6	Adaptive scrolling	31
4	Evaluation	34
4.1	Placing similar vertices on the same positions	34
4.2	Development of an interactive comparison tool	36
4.3	Design of the user study	38
4.4	Results of the user study	40
5	Conclusion and Future Work	45
	Bibliography	47

1 Introduction

Motivation These days, some companies manage and document their common workflows as digital business process models. By using these well-documented guidelines, managers can systematically improve the execution of recurrent tasks and companies can quickly train new members of staff. Similar companies tend to have similar business processes. Also, departments of the same company may evolve to use similar but different business processes. Merging these organizational units requires merging of their respective business processes. Since the process models are stored digitally in process repositories, we want to leverage modern algorithms to help with this task.

As an example for the use of process models, de Moor and Delugach [MD06] did a detailed study of a small-sized software development group – about 15 persons – that develops and maintains aerospace software. They gathered process models from official documents and interviews with a key manager familiar with both, the published process and actual practice, and proceeded to calculate the differences. In a meeting with a manager for two different projects, they reviewed the calculated differences. One such difference was that change requests were evaluated by a software lead in one project but by a whole team in another. Identifying and discussing the differences helped the company find ways to improve their practice, work regulations and process quality.

Another area of application for business process modeling and comparison is the evaluation of commercial off-the-shelf (COTS) software. An entrepreneur will usually conduct a feasibility study before settling for a specific COTS software. For taking this important decision they match the existing processes of the company to the processes imposed by the capabilities of suitable COTS software. The research project *Komplex-e*¹ concerns itself with solving this efficiently.

Lastly, you can think of business processes as graphs with nodes and edges. It is a topic of ongoing research to draw such graphs in a useful manner. Most of the results for drawing business processes can be applied in many other fields where graphs are used and drawn, and vice versa. Therefore, we can build on a great repository of research in the field of graph drawing. If it can be modeled as a graph, it can be compared like a graph. On the other hand, the same methods helping experts with comparing business processes can be used to compare other types of information.

However, without any automatic aids the visual comparison of graphs can be a cumbersome task. An example like Figure 1.1 demonstrates the problem. The two graphs have been drawn with the same algorithm. For one drawing the algorithm has been given a slightly differently ordered input and already produced a very different drawing. The

¹The results of *Komplex-e* are yet to be published, meanwhile information can be found at <http://komplex-e.de/>.

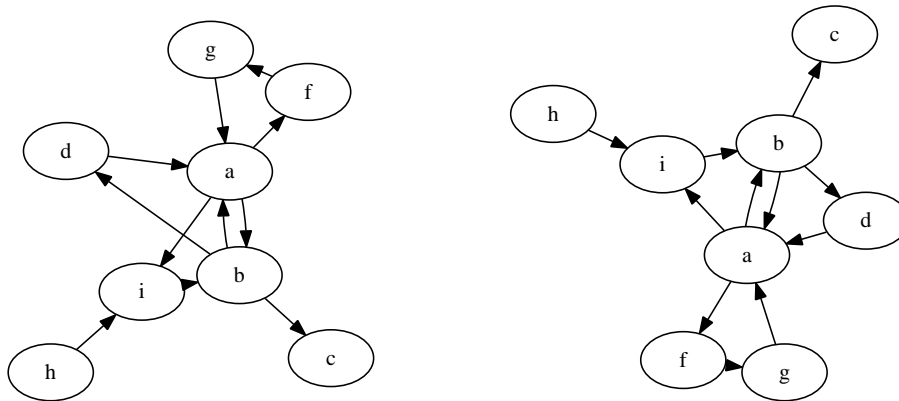


Fig. 1.1: An example where visual comparison of two graphs is hard. Both drawings show the same graph only arranged differently.

interested reader might take the time to check that the two graphs are in fact exactly the same.

Other means of documentation like plaintext or wiki hypertext sometimes offer more flexibility than the comparably rigid structure of a graph and machine learning algorithms are improving to make intelligent document comparisons feasible. Visual comparison aids like in Figure 1.2 have been available for textual information for some time. However, an analysis with graph-based algorithms comparisons might be more exact and efficient in case the information is already available in a graph-like format and could be considered as an alternative in other cases. The advances in graph theory have made a wide range of algorithms readily available. Problems when comparing results of sophisticated graph algorithms might deter a potential user from an otherwise useful approach. We strive make validating and checking the results more usable by developing and improving methods to assist a visual graph comparison. This way, we want to inspire uses of graph theory in all kinds of applications.

Related work Finding corresponding processes and quantifying their differences was already described by Dijkman et al. [DDV⁺11]. They focus on querying a repository of business processes for those that are similar to a given query. They also mention that their metric can be used to indicate the total cost of merging two companies. However, at some place in the merger human oversight is probably still required to decide how to resolve differences in process models. A similarity metric alone will not help with this, rather we want to create a diagram of the two similar business processes.

In 2015 a process model matching contest was conducted, which compared twelve different algorithms for comparing business processes and calculating matchings. In their related report, Antunes et al. [ABB⁺15] confirm a “vivid process matching community that is interested in an experimental evaluation of the developed techniques to better understand its pros and cons” but also acknowledge that there is still large room for improvement for methods that do find many relationships while at the same time not

Line 290:	Line 290:
* [http://www.utm.edu/departments/math/graph/ Graph theory tutorial]	* [http://www.utm.edu/departments/math/graph/ Graph theory tutorial]
* [http://www.gfredericks.com/main/sandbox/graphs A searchable database of small connected graphs]	* [http://www.gfredericks.com/main/sandbox/graphs A searchable database of small connected graphs]
- * {{Wayback date=20060206155001 lurl=http://www.nd.edu/~networks/gallery.htm title=Image gallery: graphs }}	+ * {{webarchive lurl=https://web.archive.org/web/20060206155001/http://www.nd.edu/~networks/gallery.htm date=February 6, 2006 title=Image gallery: graphs }}
* [http://www.babelgraph.org/links.html Concise, annotated list of graph theory resources for researchers]	* [http://www.babelgraph.org/links.html Concise, annotated list of graph theory resources for researchers]
* [http://www.kde.org/applications/education/rocs/ rocs] — a graph theory IDE	* [http://www.kde.org/applications/education/rocs/ rocs] — a graph theory IDE

Fig. 1.2: An example of visual comparison aids available for textual information. The changed text is highlighted and unmodified sections are hidden. Screenshot from the wiki-based online encyclopedia *Wikipedia*, https://en.wikipedia.org/w/index.php?title=Graph_theory&diff=prev&oldid=750169746.

finding unwanted relationships.

A survey conducted by Gleicher et al. [GAW⁺11] does not only list many systems for visual comparison of complex information, but also provides a general taxonomy of visual designs splitting them into three categories: Juxtaposition, displaying the subjects next to each other, either in time or in space, superposition or overlay and explicit representation of the relationships. They also emphasize the importance of user interaction.

As a foundation for doing the layout for the business process graphs we use the well-known algorithm by Sugiyama et al. [STT81] for hierarchical drawings and variants thereof.

When it comes to visually comparing graphs in particular, there are some more recent works of interest. Some systems receive two input graphs and a list of node similarities or a matching. For example, Holten and van Wijk [HW08] compare two hierarchies – that is, trees – using a juxtaposition. However, trees can be rearranged rather freely compared to drawings of business processes, which have to roughly maintain the order of steps.

Schreiber [Sch03] describes a method to compare graphs side by side and also bringing comparable vertices to the same horizontal position by merging them to a common vertex for some drawing steps. Their approach is strongly catered to needs of biologists and the drawing of metabolic pathways, which are a series of chemical reactions.

Andrews et al. [AWW09] have already developed a tool for comparing business processes that closely matches our requirements. In their side by side view they use a merged graph’s layout to adjust positions, whereas we adjust the individual layouts. Their method was implemented and used as a reference to evaluate our ideas.

Our contribution We have set ourselves the goal to draw a diagram of two processes in a way that provides significant improvements for business process modeling experts over just comparing two traditional drawings side by side. While a similarity metric might

make for a good first indicator for the extent of differences between processes, the actual cost of switching to another process can be estimated more exactly by examining differences in detail. We want to provide key decision makers with means of revealing those differences effectively. For this we developed an algorithm based on finding a maximum independent set of a permutation graph. This algorithm uses provided relationships between parts of a business process and adjusts existing visualizations of process models. In the basic usage, nodes are spread apart in unmatched parts of the drawing to bring related parts closer together. In a another usage, the results of the algorithm are used to adjust scroll speeds when interacting with a side by side view of two processes. The adjustment leads to related parts showing up at the same time.

To test the theoretical results in conditions close to real-world usage, we implemented an interactive tool to annotate graphs with similarity information and to compare them. We conducted a small user study to get some insights into perceived usability and helpfulness of the methods.

Structure The structure of the thesis is as follows. The next Chapter 2 gives a short introduction to the basic concepts and terminology. The algorithm developed for adjusting graph drawings for comparison itself is discussed in detail in Chapter 3. Chapter 4 describes an alternative algorithm that aids comparison and also how the methods prove themselves in a user study. To conclude, we give a summary of our results and final remarks in Chapter 5.

2 Graphs and Business Processes

Before describing the methods to aid a visual comparison of business processes, this chapter introduces the most important concepts, which build the foundation for later chapters. There are some key elements when visually comparing business process flowcharts. The theoretical foundations are graphs with labels and graph layouts. They are used as data structure in intermediate steps of the algorithms, as abstraction for the various kinds of data structures, and as a basis for models of business processes. For the application in the domain of business processes we look into event-driven process chains, which are used for modeling business processes. We are also going to show how these components play together and also define terms for a precise description in the following chapters.

2.1 Graphs

Drawing business process flowcharts is usually based on algorithms for drawing graphs.

Definition 2.1. A (*directed*) graph G is a tuple (V, E) of two sets, with $E \subset V \times V$. $v \in V$ are called *vertices* and $e \in E$ *edges*.

Graphs are the concept behind flowcharts and describe connections between vertices called edges. While there are undirected graphs, too, for flowcharts only directed graphs are of interest, where an edge $e = (v_H, v_T)$ is *directed* from its *head* v_H to its *tail* v_T . It is generally implied that there are no *loops*, that is, $v_H \neq v_T$ for all $e = (v_H, v_T) \in E$.

Here is an example of a graph G_E :

$$\begin{aligned}G_E &= (V_E, E_E) \\V_E &= \{0, 1, 2, 3, 4, 5, 6, 7, 8\} \\E_E &= \{(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (2, 6), (6, 7), (7, 8)\}\end{aligned}$$

Here, the vertices are numbers like 0, 1 and 2 and the edges are pairs like (0,1) or (1,2). We will use this graph as an example throughout this chapter.

To make graphs more useful, we assign *labels* to the vertices. We denote this using a labeling function $l_V: V \rightarrow S$ mapping vertices to strings, for example, with $S = \{\text{a} - \text{z}\}^*$. This labeling function is not necessarily injective, there could be two vertices $v_1, v_2 \in V$ with $l_V(v_1) = l_V(v_2)$. Applying this to our example G_E , we could use l_V with $l_V(0) = \text{hunger}$, $l_V(1) = \text{bake}$, $l_V(2) = \text{xor}$, $l_V(3) = \text{black}$, $l_V(4) = \text{sob}$, $l_V(5) = \text{brown}$, $l_V(6) = \text{eat}$.

While the definition of a graph as a tuple of two sets is necessary to formally define a graph's properties, for us humans a graph is intuitively described by a drawing. Such

a drawing depicts the vertices as boxes or circles and the edges as lines between them. Additionally, the edges's tails are denoted by an arrowhead. You can find examples for a drawing of the graph G_E in Figure 2.1a.

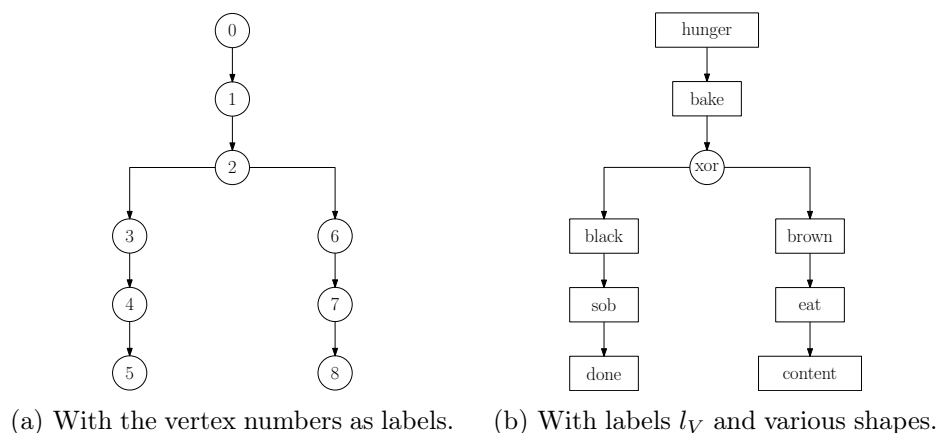


Fig. 2.1: Two drawings of the example graph G_E .

There are many variations of such drawings often including colors and different shapes as seen in Figure 2.1b. But we are mostly concerned about the positions of the vertices on the drawing pane.

Vertex colors or shapes in a graph drawing are usually assigned using domain knowledge and aesthetic considerations, but the vertex positions may be automatically assigned by a layout algorithm. The result of a layout algorithm is a layout which positions the vertices on a drawing pane.

Definition 2.2. A *layout* of a graph $G = (V, E)$ is a function $L_V: V \rightarrow \mathbb{N} \times \mathbb{N}$.

An *edge layout* is a layout L_V and a function $L_E: E \rightarrow (\mathbb{N} \times \mathbb{N})^*$ with $L_E((v_1, v_2)) = L_V(v_1), \dots, L_V(v_2)$ for all $(v_1, v_2) \in E$.

Some algorithms additionally assign paths to the edges. Instead of connecting two vertices with a straight line, edges may follow a sequence of points or even a curve to avoid areas occupied by vertex shapes. It is also sometimes desirable to draw edges using only or mostly horizontal and vertical segments, known as *orthogonal drawing*. This can be achieved using edge paths, too. To keep the edge drawings useful, they should still start and end at the positions of their head and tail vertices.

Graphs are an abstract concept and they are used to describe all kinds of information like subway networks with stations and tracks, genealogy trees with family members and their relationships, computer networks with servers and links or social networks with users and so-called friendships. Additionally they are a handy tool in computation, since algorithms are available to solve many kinds of problems. These properties are utilized in the next sections and chapters.

2.2 Business processes and event-driven process chains

In our work we used the domain of *business processes* to apply our graph algorithms to. Hansen and Neumann [HN01] give the following definition for business processes:

“A business process consists of linked activities, which are executed in a particular sequence for reaching a certain goal. Those activities can be started and executed in parallel or sequentially.”

Examples of such processes would be handling a request for time off or the purchase of goods in a trading company. An example of an activity – also known as task or step – would be calling a client. Processes and their activities are started by events, for example, when a form is submitted.

Business process modeling forms the first step when developing business-oriented software. Existing processes are analyzed and used as a basis for a new system.

When a degree of automation is introduced into a business process, it is also known as workflow. The workflows are then depicted as flowcharts. A flowchart usually exhibits graph-like characteristics, rendering events and activities as nodes and possible transitions as edges.

For the illustration of business processes we use *event-driven process chains* (EPCs). They provide a visual language that defines possible elements of a flowchart. As such, they may consist of the following elements:

Event is depicted as a hexagon-shaped node.

Function is depicted as a rounded rectangle node and they relate to activities in the process.

Logical connectors are depicted as circular nodes. They indicate a parallel execution (\wedge) or a choice between multiple execution paths (\vee and XOR).

Arcs are the edges connecting the above three types of nodes.

Instead of “XOR” we sometimes just write the symbol “ \times ”. There exist some more elements in EPCs, for example organization units, however they are not relevant for our work, as they can either be drawn and placed very similar to the other types or do not fit into our graph model very well.

Figure 2.2 is an example for an EPC.

There is a start event “hunger” on top which triggers the first action “bake”. After the baking is completed there is an XOR decision node. The workflow continues depending on which event occurs: Is the pie “black” or “brown”? If the pie is black, the next activity is “sob” until the final event “done” arises. If the pie is brown the process follows up with the activity eat, until the final event “content” triggers. As you might guess, this EPC describes the process of making and consuming pie. Notice that the XOR logical connector forces a decision for exactly one event, so you cannot have a dark-brownish cake and eat a bit before giving up and starting to sob.

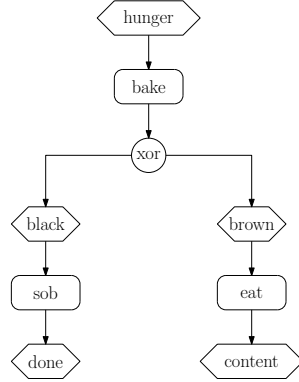


Fig. 2.2: Example for an EPC, the process of making and consuming pie.

To get a more formal grasp on EPCs we use the definition provided by W. M. P. van der Aalst [Aal99]:

Definition 2.3. An *event-driven process chain* (EPC) is a five-tuple (E, F, C, T, A) with:

- E is a finite set of events,
- F is a finite set of functions,
- C is a finite set of logical connectors,
- $T: C \rightarrow \{\wedge, \text{XOR}, \vee\}$ is a function which maps each connector onto a connector type,
- $A \subseteq (E \times F) \cup (F \times E) \cup (E \times C) \cup (C \times E) \cup (F \times C) \cup (C \times F) \cup (C \times C)$ is a set of arcs.

In the example from Figure 2.2 we would have: $E = \{\text{hunger}, \text{black}, \text{brown}\}$, $F = \{\text{bake}, \text{sob}, \text{eat}\}$, $C = \{\text{xor}\}$, $T(\text{xor}) = \text{XOR}$ and $A = \{(\text{hunger}, \text{bake}), (\text{bake}, \text{xor}), (\text{xor}, \text{black}), (\text{black}, \text{sob}), (\text{sob}, \text{done}), (\text{xor}, \text{brown}), (\text{brown}, \text{eat}), (\text{eat}, \text{content})\}$

Theorem 2.4. An EPC (E, F, C, T, A) induces a graph $G = (E \cup F \cup C, A)$.

Proof. We have to show that

$$A \subseteq (E \cup F \cup C) \times (E \cup F \cup C) \quad (*)$$

We observe that $(*)$ is true for all of the subsets $(E \times F)$, $(F \times E)$, $(E \times C)$, $(C \times E)$, $(F \times C)$, $(C \times F)$ and $(C \times C)$ of A . \square

The example EPC from Figure 2.2 induces the exemplary graph G_E from Section 2.1.

There are other methods for depicting business process flowcharts, such as Business Process Model and Notation (BPMN) or UML activity diagrams. They expose similar concepts and an application of our results to these methods should in principle be possible. However, BPMN also introduces “swim lanes”, that is, partitions of the drawing

pane into parallel swaths for each actor. This is yet another layout constraint, leading to more complex considerations and thus not covered here. An example for a business process with swim lanes can be found in Figure 2.3.

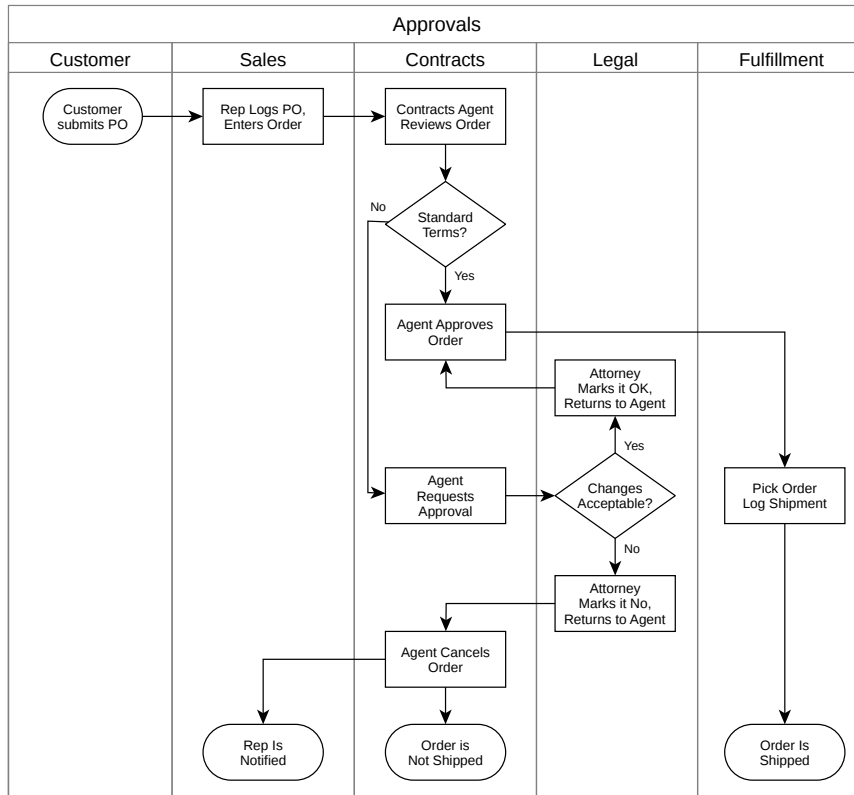


Fig. 2.3: A flow chart of a hypothetical business process with swim lanes. Here, they are vertical sections for the customer, the sales department and so on. Source: <https://commons.wikimedia.org/wiki/File:Approvals.svg>.

2.3 Finding or specifying similarities

A useful approach when comparing business processes is not to focus on the possibly many differences, but on the similarities first. Comparable processes usually have some functions in common. In our example with a recipe as process, this could be steps like “bake” or “eat” that occur while preparing and consuming a meal, even if other steps differ. In a more business-oriented setting, those would be functions like posting a goods receipt, confirming a payment, checking delivery instructions or entering data. The same is true for events. For example, the occurrence of “hunger” would be a common starting point for two recipes. Even though logical connectors are widespread, they are hard to match, since for example a XOR connector is unrelated to most other XOR connectors in another process.

Those similarities are usually described with a value between 0 and 1, where 0 means no similarity and 1 means semantically equivalent. A set of similarities is sometimes called *matching*, *alignment* or *correspondences*. Sometimes a whole group of n elements matches another group of m elements and they form an $n : m$ matching. The most common case of matching is two elements in an $1 : 1$ matching with a similarity value of 1. For the methods described in the following chapter this latter matching is the only one of interest, other matchings can be reduced to this case. We can apply a threshold value to the similarities to decide between 0 and 1 and unfold $n : m$ matchings into nm single $1 : 1$ matchings. A matching with a value of 0 is usually implied when no other matching is given for a set of elements. To clarify which kind of matching is used for adjusting the graph drawings for comparison, we call the resulting $1 : 1$ relationships between similar functions and events *constraints*.

Definition 2.5. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be graphs. A *constraint* between G_1 and G_2 is a two-tuple (v, v') with $v \in E_1$ and $v' \in E_2$.

In the case that G_1 and G_2 are graphs induced by business processes with $G_1 = (E'_1 \cup F_1 \cup C_1, A_1)$ and $G_2 = (E'_2 \cup F_2 \cup C_2, A_2)$, we restrict this to $v \in E'_1 \cup F_1$ and $v' \in E'_2 \cup F_2$.

The graph nodes a and b are the equal or similar elements of the graphs or processes, as determined by experts or automatically. In the context of business processes, the task of finding these similarities automatically is called *process model matching*. It constitutes a vast research topic that cannot be surveyed here. It is related to *ontology alignment*, which more generally describes finding similarities in concepts. We developed a preprocessor that will mark functions and events as similar if they have equal labels.

2.4 Sugiyama's layout algorithm

Sugiyama et al. [STT81] proposed a framework for layered graph drawing, which was henceforth also known as Sugiyama-style graph drawing. It is based around the idea of drawing graphs partitioned into layers. In the process, each vertex is assigned a vertical position based on its layer and a horizontal ordering reducing edge crossings. The resulting drawing should align as many directed edges as possible from top to bottom. It is furthermore desirable to have few and narrow layers for a good use of drawing space. In regard to edges, we want them drawn short and with only few bends and crossings. Note that these considerations are usually mutually exclusive: To get a very narrow drawing, we could place all vertices on one straight vertical line. But this would cause the resulting drawing to be very high and the edges would probably need many bends to route around the vertices. For this reason, different techniques were developed to balance the requirements. However, all of these techniques fit in a framework of five steps to draw a graph in a layered drawing: Breaking of cycles, assigning of layers to vertices, ordering the vertices in each layer, fixing the horizontal position of vertices and drawing the edges.

Cycle breaking Layered graph drawing is only applied to directed acyclic graphs. But this can be extended to general directed graphs by breaking all cycles. If we flip an

edge which is part of one or many cycles, these cycles are then paths. We can continue flipping edges until no cycles remain. However, any edges flipped here are later drawn upward against the general direction of edges. To get a consistent drawing we want to keep the number of upward edges in the layered graph drawing low, but the problem of selecting a minimum set of edges to flip – a *minimum feedback arc set* – is NP-complete. Fortunately, there are fast approximation algorithms, most of which select edges greedily by different criteria.

Layer assignment The next step is to assigning the vertices to layers. We want to end up with a partition $V_0, V_1, \dots, V_n = V$ of a graph $G = (V, E)$, that is, $V_i \cap V_k = \emptyset$ if $i \neq k$. Such a partition is called *hierarchy* if all edges $(v_i, v_k) \in E$ with $v_i \in V_i$ and $v_k \in V_k$ have $i < k$. If all edges also satisfy $i + 1 = k$ the hierarchy is called *proper*. The subsets V_i are called *layers*. It is not hard to see that cycles prevent us from constructing such a hierarchy. A proper hierarchy can be created from a hierarchy by inserting dummy nodes on edges when they cross more than one layer.

There is a very simple linear time algorithm to assign all vertices to layers. We layer V_0 to the set of source vertices, that is, all vertices that have no incoming edges. We then assign a vertex v to layer V_i where i is the length of a longest path from a source to this vertex. This already minimizes the number of layers needed.

Further optimizations are possible which reduce the total edge length. One common example is that a source vertex v is connected only to a vertex v_k in a very low level V_k . If we then set $v \in V_{k-1}$ instead of $v \in V_0$ we would save the height of these $k - 1$ layers in edge length.

Other variations of the layer assignment step limit the number of vertices per layer and thus the width of the drawing, while accepting a higher number of layers. We did not find any examples of very wide layers in our test set of business processes. We conjecture that drawing a vertices on a different layers which would belong on the same layer only to save space might decrease usability. Therefore we did not further look into such optimizations.

Vertex ordering After the vertices are distributed to layers the next step reduces edge crossings. The vertices are first ordered arbitrarily in a layer, leading to many edge crossings. The untangling is a combinatoric problem, so an optimal algorithm leads to a high runtime. A faster heuristic is the *barycentric method*. The layers are alternately processed from top to bottom and from bottom to top. The processing from top to bottom is repeated for a fixed number of times or until no changes were made in one run. For each layer the crossings between the next layer are reduced. Each vertex from the neighboring layer is placed on the average horizontal coordinate of its peers in the current layer. Top and bottom layer switch roles when processing from bottom to top. In the end, the number of edge crossings is sufficiently low in an acceptable runtime.

Horizontal positioning After assigning layers to the vertices and determining their relative order in their respective layer, the vertices have to be placed on the drawing

pane. While their vertical position is directly correlated to the layer, the horizontal position must be carefully computed to reduce the number of edge bends. The vertex positioning may induce conflicts leading to either a wider drawing or more edge bends. An optimal algorithm to solve the problem uses quadratic programming and minimizes the distance to a straight path between the edge endpoints. However, it is slow for larger problem instances and may lead to wide drawings. A heuristic method, called *priority layout method* works similar to the barycentric method and processes the drawing layer by layer upwards and downwards. In each step the vertices of a layer are processed in the order of their priority. Dummy vertices have the highest priority followed by vertices connected to many others. The positions of high priority vertices are adjusted first to make edges vertical. Then the vertices with lower priority follow. After a fixed amount of iterations a good drawing is achieved.

Edge drawing As a final step the edges are drawn. We can use simple straight lines, orthogonal connections or more complex methods. The dummy nodes inserted previously make sure that edges do not cross through any vertices.

Fast algorithms that do almost no optimization may also deliver acceptable solutions for some problem instances. Finding an optimal solution for most sub-problems is NP-hard, but acceptable heuristics are known. Sugiyama's layout algorithm is considered a well-researched tool to draw individual processes. Also, when comparing processes it is sometimes already sufficient to draw them both in a hierarchical layout and place them side by side. If this is not enough, more measures can be applied like the ones described next chapter.

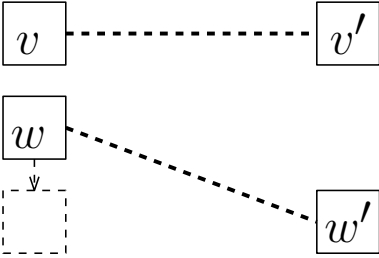
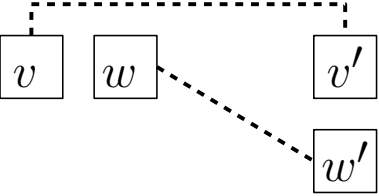
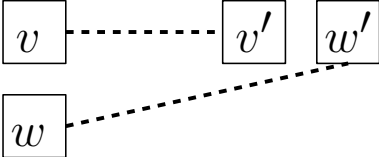
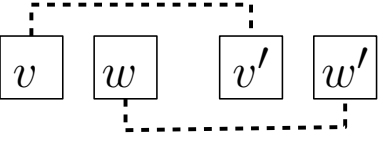
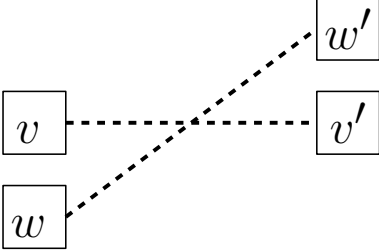
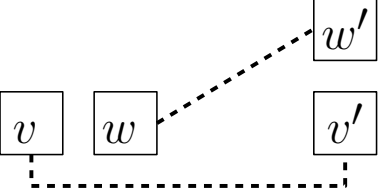
3 Placing Similar Vertices on the Same Height

When comparing business processes without a defined process, assessors look at companies or departments and use their intuition and experience to gather information on process differences. Tools like EPCs were developed to help retaining and sharing the gathered knowledge. The next step is now to compare the gathered processes. This is usually done manually, using the artifacts from the surveying step. We want to explore methods of assisting this second phase with automatically calculated visual aids. Our approach strives to improve the experience when comparing two business process flowcharts side by side. The methods do not only work on EPCs but on graphs in general. After drawing the individual graphs using for instance Sugiyama's layout algorithm from Section 2.4, we can trivially arrange the drawings side by side to compare them. However, it would be useful to include information gathered on similarities. In Section 2.3 we already described how similar or equal elements of processes might be assessed and described using constraints. To visualize a constraint (v, v') between the vertices v and v' of two graphs it would be a good idea to just draw a straight line between the representations of v and v' in the drawings. To differentiate those from the edges of the individual graphs, we can use another color and shape. For large graphs, however, it can be cumbersome to follow long constraint lines across the drawing pane. This is why we want to use the constraints in another way: To place similar parts of the graphs next to each other. This can be achieved by putting vertices with constraints on the same height, thereby making the drawn line for the constraint horizontal. In this chapter we want to describe a method of adjusting an existing layout to establish this property, at least for some constraints.

3.1 A simple adjustment to make two constraints horizontal

The method of adjusting n constraints with our method builds on a procedure to make two constraints horizontal. After drawing both graphs with Sugiyama's layout algorithm, they both are partitioned into layers V_0, V_1, \dots, V_n . Assume we are given a set of constraints $C = \{(v, v'), (w, w'), \dots\}$ with $v \in V_{L(v)}$ of the first graph and $v' \in V_{L(v')}$ of the second graph. This defines a *layer assignment* L mapping each vertex to the number of the layer in which it is contained.

We can bring v and v' to the same height by shifting the whole drawing of the second graph $L(v) - L(v')$ layers downwards. For example, v' was in layer $V_{L(v')}$ before and is in layer $V_{L(v')+(L(v)-L(v'))} = V_{L(v)}$ after the shift and we have our desired property of v and v' being on the same height, $v, v' \in V_{L(v)}$.

	$L(v) < L(w)$	$L(v) = L(w)$
$L(v') < L(w')$	 <p>Simple adjustment possible, we can move w or w' down, depending on whether $L(w) < L(w')$ or not.</p>	 <p>Simple adjustment impossible, since we would have to merge or split layers.</p>
$L(v') = L(w')$	 <p>Simple adjustment impossible, since we would have to merge or split layers.</p>	 <p>Simple adjustment unnecessary, the vertices w and w' are already on the same height.</p>
$L(v') > L(w')$	 <p>Simple adjustment impossible, since the constraints cross.</p>	 <p>Simple adjustment impossible, since we would have to merge or split layers.</p>

Tab. 3.1: All cases when trying a simple adjustment of two constraints.

After the shift, w and w' are most likely still on different layers, let's call them $w \in V_{L(w)}$ and $w' \in V_{L(w')}$. Without loss of generality assume that the constraints are ordered by the height of the layer of their vertex in the first graph, that is, $L(v) \leq L(w)$ or v is above w in the drawing. Flip the constraints otherwise.

If we then want to continue and bring w and w' to the same height, without at the same time moving v and v' to different heights again, we can do what we call a *simple adjustment*. For the adjustment to work, v' has to be strictly above w' in the shifted drawing, $L(v') < L(w')$, and v and w have to be on different layers, $L(v) \neq L(w)$. Then look at the height difference of the constraint between w and w' , $L(w) - L(w')$. If $L(w) < L(w')$ insert $L(w') - L(w)$ layers into the drawings of the first graph just before $V_{L(w)}$. If $L(w') < L(w)$ insert $L(w) - L(w')$ layers into the drawings of the second graph just before $V_{L(w')}$. After this operation, w and w' , too, are on the same height.

What about the cases when this adjustment does not work? First the case where v and w are on the same layer: In this case we are essentially asking for v, v', w, w' to all be on the same layer: After the first shift, we already established $L(v) = L(v')$ and we have $L(v) = L(w)$ by definition. The adjustment should make $L(w) = L(w')$. These equalities come together when v' and w' are on the same layer, $L(v') = L(w')$, and we already have $L(w) = L(v) = L(v') = L(w')$, so no adjustment is needed. On the other hand, when v' and w' are not on the same layer, $L(v') \neq L(w')$, our adjustment would have to put v' and w' in the same layer or split v and w into different layers. This is not possible by just inserting new, empty layers, we would have to make further adjustments to accommodate for the other elements of the drawings.

Secondly, v' may not be strictly above w' . If they are on the same layer, we end up with the mirrored condition of the last paragraph, so again either is $L(v) = L(w)$ and there is no need for adjustment, or we would have to split or merge layers. The other case, v' is strictly below w' , means that the lines of the constraints (v, v') and (w, w') cross in the drawing. Then again, simple inserting of layers on one side is no longer possible and we would have to rearrange the drawing in more complex ways to accommodate both $L(v) = L(v')$ and $L(w) = L(w')$ at the same time while still maintaining a consistent hierarchical drawing.

All in all the cases in Table 3.1 are distinguishable. Note that $L(v) > L(w)$ does not appear in any of the cases, because we initially assumed without loss of generality that $L(v) \leq L(w)$. We call the constraint (w, w') in *conflict* with (v, v') if we cannot do a simple adjustment. We call the constraint (w, w') *redundant* to (v, v') if $L(v) = L(w)$ and $L(v') = L(w')$.

3.2 Conflicts between n constraints

With the method described until now, a drawing can be adjusted to level out two constraints. We want to build on the concepts introduced in the last section and develop an algorithm for n constraints in a set of constraints $C = \{(v, v'), (w, w'), \dots\}$. Figure 3.1 shows an example of two graphs with eight constraints, three of which are already horizontal. In the last section $L(v) = L(v')$ was established after the first shifting of the

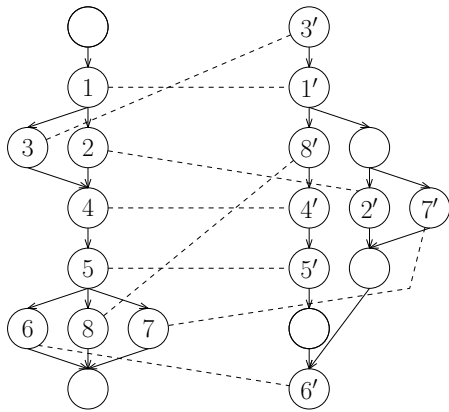


Fig. 3.1: Two graphs with constraints as dashed lines

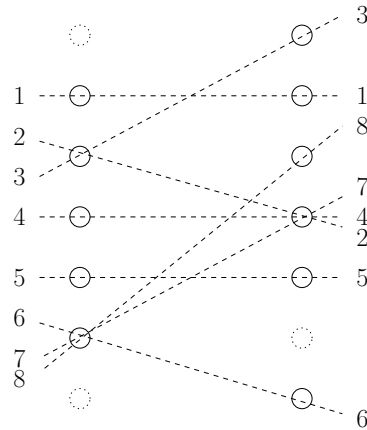


Fig. 3.2: Constraints between layers define a permutation

second graph. However, the cases outlined in Table 3.1 only ever use the relative positions of v and w and of v' and w' and make no assumptions about the relationship between v and v' . A simple adjustment will only insert new layers and thus keep the relative order of vertices in the drawing. Consequently, if two constraints have the properties $L(v') < L(w')$ and $L(v) < L(w)$ after the shift to make (v, v') horizontal – indicating that we can do the simple adjustment – they already had this property before the shift. Equally, redundancy and conflicts persist irrespective of simple adjustments. In further consequence, if there are two conflicting constraints in the initial drawing, no sequence of simple adjustments can resolve this conflict.

It is therefore sound to introduce a *conflict graph* $G_C = (C, E_C)$. The set of vertices of the conflict is the set of constraints C . The set of edges is the set of pairs of constraints in conflict. Figure 3.4 depicts the conflict graph for the constraints from Figure 3.1. We labeled the vertices for the constraints in the drawing of G_C with their first vertex for brevity. In the example the constraint $(1, 1')$ crosses the constraint $(3, 3')$, since 1 is strictly above 3 but $1'$ is strictly below $3'$. The constraints $(3, 3')$ and $(2, 2')$ do not cross but start in the same layer in the first, left graph so they, too, are in conflict. We cannot draw them both horizontal without merging layers in the second, right graph. The constraints $(2, 2')$ and $(7, 7')$ are conflicting because they end in the same layer in the second graph. There are many pairs of constraints that are not in conflict, for example, $(4, 4')$ and $(5, 5')$ are not since they are parallel. Here, 4 is strictly above 5 and $4'$ is strictly above $5'$, so we could do the simple adjustment, were they not horizontal already.

3.3 Selecting as many constraints as possible

Because of the conflicts between constraints, they cannot all be drawn horizontal using simple adjustments. We can, however, pick as many pairwise non-conflicting constraints as possible. In terms of the conflict graph, we want to find a *maximum independent set*.

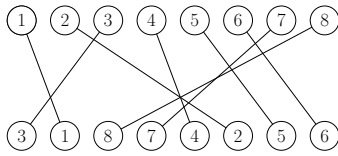


Fig. 3.3: Matching diagram of the permutation 3, 1, 8, 7, 4, 2, 5, 6

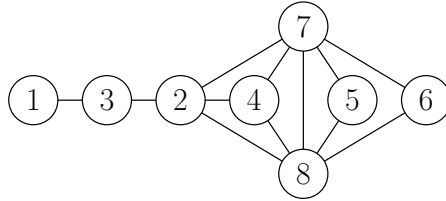


Fig. 3.4: Permutation graph of the permutation 3, 1, 8, 7, 4, 2, 5, 6. Also conflict graph of the example constraints from Figure 3.1

An *independent set* of a graph is a subset of its vertices, no two of which are adjacent, that is, no edge is connecting the two. The maximum independent set is a set with the highest cardinality of all possible independent sets. Finding a maximum independent set is NP-hard for general graphs, but we will show that conflict graphs from constraints in drawings belong to a class of graphs for which the problem is solvable in polynomial time.

A *permutation* π of integers $1, 2, \dots, n$ is a bijection assigning each number exactly one other number. A permutation of integers can be fully specified by the sequence $\pi(1), \pi(2), \dots, \pi(n)$. The inverse π^{-1} of the bijection assigns each number its position in this sequence. A pair of numbers a, b with $a < b$ and $\pi^{-1}(a) > \pi^{-1}(b)$ is called *inversion*. The *inversion graph* or *permutation graph* $G[\pi] = (V, E)$ of the permutation π is the graph with the numbers as set $V = \{1, 2, \dots, n\}$ of vertices and an edge between two numbers if they are inverted in π , so $(a, b) \in E$ if and only if $a < b$ and $\pi^{-1}(a) > \pi^{-1}(b)$. All graphs that are isomorphic to a graph $G[\pi]$ are called *permutation graphs*. For example, the permutation 3, 1, 8, 7, 4, 2, 5, 6 has $\pi(1) = 3, \pi(2) = 1, \pi(3) = 8, \dots, \pi(8) = 6$ and the permutation graph in Figure 3.4. Permutation graphs were introduced by Even et al. [EPL72] motivated by problems in memory allocation and circuit layout and have since found their way into textbooks, like the one from M. C. Golumbic [Gol04].

Theorem 3.1. *The conflict graph of constraints between two Sugiyama-style drawings is a permutation graph.*

Proof. When using only simple adjustments, all that matters about the constraints are the layers of their vertices and not their horizontal positions because the vertices are never moved independently. We construct a diagram where we draw two columns of circles, one column for each input graph, like in Figure 3.2. We then represent the layers with small circles in the order of their vertical arrangement in the drawing. As a next step, we draw a line for each constraint connecting the pair of circles belonging to the layers of its vertices. However, we draw the constraint lines just a little longer, in such a way that all the lines of each layer cross in the center of its circle, but not too long, so that we do not create any other crossings with unrelated lines. Notice that for sets of pairwise redundant constraints, we can only draw one line. This means that all constraint lines linked to a layer are connected to different circles in the other column, so they have

different slopes and cross in the circle. All in all we have constructed a diagram where constraint lines cross if and only if the corresponding constraints are in conflict. If we now number the endpoints of the lines on the left side with $1, 2, \dots, n$ from top to bottom and repeat each number on the other endpoint of the line, we get a new sequence on the right side. This sequence is a *permutation* of the numbers $1, 2, \dots, n$. In the resulting diagram the i th and the k th constraint lines cross if and only if i and k are reversed in the permutation. This means that the inversions of the permutation, the intersection graph of the constraint lines in the constructed diagrams and the conflict graph of the constraints are equivalent. \square

If we construct such a diagram for the constraints in Figure 3.1 we end up with Figure 3.2. In the right column one can read off the permutation $3, 1, 8, 7, 4, 2, 5, 6$, whose permutation graph we have already seen in Figure 3.4, so this is the conflict graph of the constraints, too. For the sake of the example the nodes and constraints of Figure 3.1 are conveniently named to correspond with the numbers of the permutation.

The construction of the diagram used in our proof is similar to *matching diagrams* or *permutation diagrams* defined in literature. A matching diagram is constructed from a permutation by writing $1, 2, \dots, n$ in one line and the numbers $\pi(1), \pi(2), \dots, \pi(n)$ in the next line. Then the equal numbers are connected. For the permutation $3, 1, 8, 7, 4, 2, 5, 6$ used across the examples we can obtain the matching diagram in Figure 3.3.

3.4 Finding a maximum independent set of a permutation graph

In the last section it was shown that the problem of selecting a maximum number of constraints is reducible to finding a maximum independent set of permutation graphs. This section is about efficient algorithms for this task.

Our initial idea was to solve this using a graph algorithm. We use the permutation's integers $1, 2, \dots, n$ as vertices of a graph. We also add two special vertices s and t , where s has an outgoing edge to every other vertex and t has an edge from other vertex, that is, two vertices u and v are connected by a directed edge (u, v) if $u = s$ or $v = t$. Also we add an edge (u, v) between u and v if there is no inversion between u and v , meaning they are a smaller number u followed by a larger number v in the sequence, so $u, v \notin \{s, t\}$, $u < v$ and $\pi^{-1}(u) < \pi^{-1}(v)$. No other edges are added.

The resulting graph is acyclic, since the condition $\pi^{-1}(u) < \pi^{-1}(v)$ makes sure edges from any number are always oriented to other numbers later in the sequence of the permutation, with s and t as sentinel vertices for a first and a last element, respectively. This means that a path from s to t can be interpreted as a subsequence of the sequence of the permutation. The vertices of the path, excluding the sentinel vertices s and t are the elements of the subsequence. For example, a path with edges $(s, u), (u, v), (v, w), (w, t)$ would denote the subsequence uvw .

Additionally, the condition $u < v$ makes sure that the edges go from a smaller number to a higher number with s and t as sentinel vertices for a smallest and largest number,

respectively. This means that any path found is also an increasing subsequence.

On the other hand, all increasing subsequences can be represented as paths in the constructed graph, since the corresponding edges have been added in the construction. Any pair of consecutive numbers u, v in an increasing subsequence has the properties that $u < v$, since the subsequence is increasing, and $\pi^{-1}(u) < \pi^{-1}(v)$ since the subsequence keeps the same order as the permutation's sequence. We can also see that the length of the subsequence is the length of its corresponding path minus one.

A longest increasing subsequence is thus a longest path in the constructed directed acyclic graph. We can use a well-known algorithm for finding such a path, sometimes called *critical path method*. The algorithm works by first finding a topological ordering on the graph, in our case we can use π for this purpose, adding s and t as first and last vertex, respectively. Secondly traverse the vertices in the topological order. For s set the path length to 0. For all other vertices set the path length to the maximum of the path lengths of the vertices with edges to this vertex plus 1. In the end path length of the vertex t will be the length of a longest path in the whole graph from s to t . By backtracking which vertex contributed the individual paths lengths we can find the longest path.

Because we have to check each vertex, and all incident vertices for them, this algorithm runs in $O(|V| + |E|)$ time. The set V is $\{s, 1, \dots, n, t\}$ by construction, so its cardinality is $|V| = n + 2$. However, the cardinality of the set of edges $|E|$ is only limited by $O(n^2)$; For the identity permutation $1, 2, \dots, n$ we get the highest cardinality of $|E| = n + n + n(n - 1)/2 + 1 = (n^2 + 3n + 2)/2 = (n + 2)(n + 1)/2$, consisting of the n edges from s to all numbers, the n edges from all numbers to t , the $n(n - 1)/2$ edges between the numbers and the edge (s, t) . Ignoring the directions of the edges, the graph would be a complete graph, meaning all vertices are connected with all others. The runtime in terms of the size of the permutation's sequence n is therefore in $O(n^2)$.

Once we have found a longest increasing subsequence, we have also found a maximum independent set of the permutation graph. It is clear that not only subsequent elements u, v of an increasing subsequence have the property $\pi^{-1}(u) < \pi^{-1}(v)$ but also any pair of elements of the sequence, since the order is not changed from the order of the elements in the permutation's sequence. This also means that any longest path we find in the constructed graph is also a clique when ignoring the directions of the edges, that is, any two vertices of the path are also connected by an edge.

This transitivity suggests that there are many redundant edges in the constructed graph. If there are edges (u, v) and (v, w) , the edge (u, w) must not even be considered by our algorithm, because if (u, w) was part of a longest path, there would be no elements between u and w in the corresponding subsequence. But we could add v to make the sequence one element larger, meaning we would not have found a longest path in the first place, leading to a contradiction. In fact, Even et al. [EPL72] show how it is sufficient to find certain transitivity properties to classify a graph as permutation graph. However, it is not immediately clear how an algorithm can make use of these properties to reduce the theoretical runtime bound to something better than $O(n^2)$.

To create a fast algorithm we used prior results from Aldous and Diaconis [AD99] and Kim [Kim90] and ended up with Algorithm 1.

Algorithm 1: LongestIncreasingSubsequence(Array A , Relation $<$)

input : Array of permuted elements A and a relation $<$ indicating their original order

output: A longest increasing subsequence of A according to $<$

```

1  $piles =$  new, empty list
2 foreach  $a \in A$  do
3    $index = piles.binarySearch(\text{first with } a < \text{last added element})$ 
4   if  $index$  indicates not found then
5      $piles.append(\text{new queue with only } a \text{ on it})$ 
6   else
7      $piles[index].enqueue(a)$ 
8  $subsequence =$  new, empty list
9  $last = null$ 
10 foreach  $p \in \text{reverse}(piles)$  do
11   until  $last$  is  $null$  or  $p.peek() < last$  do
12      $p.dequeue()$ 
13    $last = p.dequeue()$ 
14    $subsequence.append(last)$ 
15 return  $\text{reverse}(subsequence)$ 

```

We want to describe Algorithm 1 and show that it finds a longest increasing subsequence. The loop starting in line 2 goes through all the elements x in the order of the input permutation sequence A and places each element on a pile. The piles are implemented as first-in-first-out queues and kept in an ordered pile list $piles$. In every iteration the first pile in this list with the property that the element x to be piled is smaller than its last added element is selected. If there is no such pile, the element x is placed on a new pile at the end of the list $piles$. This ensures a few invariants for the already added elements across the iterations of the loop. Firstly, the elements on each individual pile are ordered from large to small, with the smallest having been added last. Secondly, the elements of each individual pile appear in the order of the input list, with the first added element being the first to occur in the input list from all elements in its particular pile. Lastly, the last added elements from each of the piles in the order of $piles$ form an increasing sequence, making the binary search possible.

The second part of the algorithm from line 8 on builds an output list $subsequence$ from these piles. First it takes the firstly added element of the last pile in $piles$ to $subsequence$, which is the largest element of this pile. It then continues backwards through $piles$ and takes one element from each pile. For this it looks through its elements from the one added first, until an element smaller than the last element in $subsequence$ is found. It then adds this element to the end of $subsequence$. In the end the algorithm returns the reverse of the list $subsequence$.

As an example, suppose $A = 3, 1, 8, 7, 4, 2, 5, 6$, the sequence known from the previous

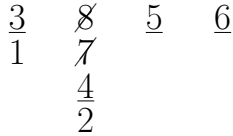


Fig. 3.5: Result of an example run of Algorithm 1 on the input sequence $A = 3, 1, 8, 7, 4, 2, 5, 6$. The piles are depicted as vertical columns with last added elements on the bottom. The elements of *subsequence* are underlined for clarity, the elements discarded in the second phase are crossed out.

examples such as Figure 3.2. The results of the algorithm along with the piles built can be found in Figure 3.5. The first phase places the numbers of the piles depicted as vertical columns with last added elements on the bottom. The first number 3 is placed on the first pile. Then 1 is placed on the same pile, since 3 is the last added element on the pile and $1 < 3$. The next number 8 has to be placed on a new pile because $8 \not< 1$. The other numbers are placed accordingly. In the second phase the algorithm starts with the first number of the last pile, 6, and puts it into *subsequence*. It then proceeds to take 5 from the penultimate pile. When it starts looking at the second pile *last* is set to 5, so the loop in line 11 dequeues and discards 8 and 7, since $8 \not< 5$ and $7 \not< 1$. The loop terminates at 4 when $4 < 5$, and 4 is added to *subsequence*. Lastly 3 is added, yielding *subsequence* = 6, 5, 4, 3 which when reversed is 3, 4, 5, 6 one of the longest increasing subsequences of $A = \underline{3}, 1, 8, 7, \underline{4}, 2, \underline{5}, \underline{6}$. Keep in mind that there may be more than one longest increasing subsequences. The example sequence 3, 1, 8, 7, 4, 2, 5, 6 has the increasing subsequences 1, 2, 5, 6, 1, 4, 5, 6 and 3, 4, 5, 6, which are all of maximum length 4. Now that we have defined some basic properties of the algorithm and the piles, it is time to proof correctness and show that the algorithm does indeed return a longest increasing subsequence.

Theorem 3.2. *The algorithm LongestIncreasingSubsequence calculates a longest increasing subsequence.*

Proof. As a first step assume that the input sequence A has a longest increasing subsequence $a_1, a_2, a_3, \dots, a_l$ of length l with $a_1 < a_2 < \dots < a_l$. If the algorithm encounters a_1 it is placed on a pile. Afterwards it will encounter a_2 . The algorithm cannot chose the pile containing a_1 , because $a_1 < a_2$. Even if a_1 is no longer the last added element on its pile and some other elements x, y, \dots, z have been added afterwards, they would have the property $z < \dots < y < x < a_1$ because that is the condition under which the elements x, y, \dots, z are placed on the pile and with $a_1 < a_2$ we get $z < a_2$. Recall that the condition in line 3 mandates $z > a_2$, so the algorithm chooses another pile. One can argue analogously that a_3 cannot be placed on neither the pile with a_1 nor the pile with a_2 , so it has to be placed on a third pile. Ultimately there must be at least l piles.

The second loop in line 10 selects one element from each pile. Having more than l piles would lead to a contradiction, because the algorithm would find an increasing subsequence even longer than our assumed longest increasing subsequence $a_1, a_2, a_3, \dots, a_l$, thus it allocates at most l piles and finds a longest increasing subsequence.

However, now that we know our returned result has the correct length, we still have to prove we find an increasing subsequence after all. When the second loop in line 10 starts all elements have been placed on l piles maintaining the invariants of the first loop. In the first iteration the last pile in *piles* is processed and *last* is *null*. This means the inner loop in line 11 is not entered and the lines after the loop dequeue an arbitrary element r_0 from the last pile, store it in *last* and add it to the result list *subsequence*. Notice that when a pile is added it has at least one element, so the dequeue operation always returns an element.

For the second loop starting in line 10 we want to maintain the following invariant: After the i th iteration *subsequence* contains an increasing subsequence r_i, r_{i-1}, \dots, r_0 of A of length i reversed and *last* is r_i . For just the single element r_0 this is trivial. Assume r_i was just added from pile $l - i$ to the *subsequence* in the last iteration i , set *last* = r_i , maintained the invariant and now start iteration $i + 1$. The inner loop removes elements from the pile $l - (i + 1)$ until the element that would be removed is $< \textit{last} = r_i$. Next *last* is set to the element r_{i+1} that stopped the loop, that is, the first element that is smaller than r_i , and added to *subsequence*. Since r_{i+1} was selected such that $r_{i+1} < r_i$ an increasing sequence is maintained. Do we always find such an element and is $r_{i+1}, r_i, r_{i-1}, \dots, r_0$ still a valid subsequence of A ? At the moment when r_i was added to pile $l - i$ there must have been an element r'_{i+1} as last added element on pile $l - (i + 1)$ with $r'_{i+1} < r_i$ or else pile $l - (i + 1)$ would have been selected by the binary search. This means, if the inner loop does not stop at another element r''_{i+1} first, it will eventually stop at element r'_{i+1} and we can use it as r_{i+1} . Because the element r'_{i+1} was already on the piles when r_i was encountered it stands before r_i in the input list A . Therefore *subsequence* is still a valid subsequence. In the other case, when the loop stops at another element r''_{i+1} before r'_{i+1} in the pile, we know that r''_{i+1} came before r'_{i+1} in A since its position before r'_{i+1} in the queue means it was encountered before r'_{i+1} while traversing A in-order. With this and our knowledge that r'_{i+1} stands before r_i we also know that r''_{i+1} came before r_i , so we use it as r_{i+1} and still maintain the invariant.

After l iterations l elements have been added to *subsequence* and by induction it contains a reversed increasing subsequence. As a last step, the algorithm returns this list reversed and thus calculates a longest increasing subsequence. \square

Notice that while the last added elements of the piles are in an increasing sequence with maximal length l , they are not necessarily a subsequence of the input sequence A . If a small element comes late in the sequence, it may be placed on a previous pile, thus violating the order of the input sequence. An example for this is the input sequence $A = 4, 2, 3, 1$, you can find the resulting piles in Figure 3.6. The numbers 4 and 2 are placed on the first pile, then 3 on the second and lastly 1 on the first pile again. In the second phase, 3 is taken from the last pile, 4 is discarded and 2 is taken from the first pile. Notice that the last added elements of each pile 1, 3 are ascending but not a subsequence of $A = 4, 2, 3, 1$, because there they appear in the order 3, 1. However, the result of the algorithm, numbers 2, 3, are a subsequence of $A = 4, \underline{2}, \underline{3}, 1$.

Our initial graph-based algorithm had a runtime of $O(n^2)$ and we suspected that it does some unnecessary work. We can now observe an improvement with the theoretical

4	3
2	3
1	3

Fig. 3.6: Result of an example run of Algorithm 1 on the input sequence $A = 4, 2, \underline{3}, 1$. The last added elements of each pile are ascending but not a subsequence of A .

upper bound of the runtime of Algorithm 1.

Theorem 3.3. *The algorithm `LongestIncreasingSubsequence` runs in $O(n \log n)$ time, with n being the size of the input array A .*

Proof. Creating the empty lists, assigning *null* and reversing *subsequence* are all at most linear time operations. So let's look at the loops. The first loop in line 2 runs exactly n times. In each iteration it can create at most one pile, so we don't end up with more than n piles. In each iteration it does a binary search in line 3. Since we can make sure that indexed dereferences in our *piles* list is a constant time operation, the list is sorted and has at most n elements, the binary search finds the required *index* in $O(\log n)$ time. Appending to the *piles* list and looking at the last added element are constant time operations, too, given apt data structures are used, so the total contribution of this loop to the runtime bound is $O(n \log n)$. The second loop in line 10 is run once for each pile. Again, the number of piles is limited by n . The inner loop in line 11 may remove every element only once and there are at most n elements. If the dequeuing and appending operations are constant time, as is the case for e.g. linked lists, this loop contributes $O(n)$ to the runtime bound. In total the runtime of the algorithm is bound by $O(n \log n)$. \square

The algorithm is mostly based on the one by Aldous and Diaconis [AD99]. However, the second loop with the collection of the results is inspired by Kim [Kim90]. Aldous and Diaconis used pointers between the piles to build the final result list, but a loop based approach seemed easier to implement. Kim's approach is to iterate based on the natural order of the elements and calculate the piles by comparing the positions of the numbers, but having a permuted list as input was more flexible in our use-case. Also it overlaps nicely with the sorting algorithm *patience sort*, where the second phase of the algorithm is replaced by a k -way merge: It repeatedly takes the smallest element from the piles to form a sorted list as output. The shared pile-building phase is familiar for those already knowing the sorting algorithm.

This section resulted in a fast and easy to implement algorithm for finding a longest increasing subsequence and thus a maximum independent set of a permutation graph, which we can use to select a set of non-conflicting constraints. What is still missing is how to use these results to adjust the layout.

3.5 Calculating offset lists for adjusting the layout

Recall that our initial problem was given a set of constraints $C = \{(v, v'), (w, w'), \dots\}$ and a layer assignment L find a way to adjust this assignment to draw as many constraints

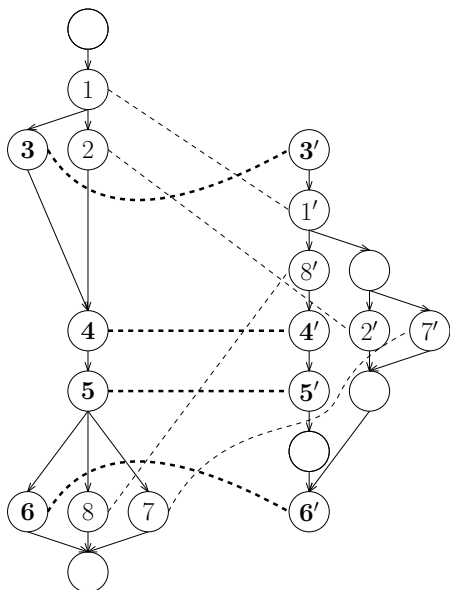


Fig. 3.7: The two graphs from Figure 3.1 after adjustment

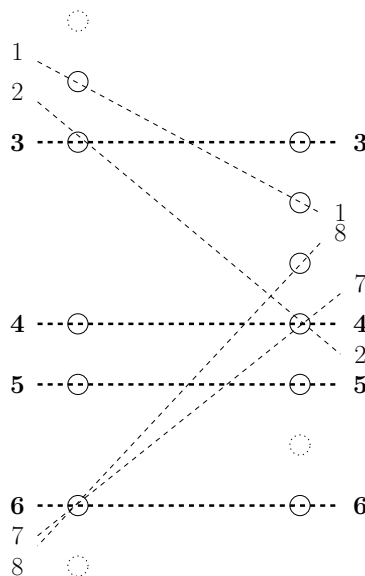


Fig. 3.8: The constraints from Figure 3.2 after adjustment

as possible horizontally, yielding a new assignment L_A . For this start at the top and the repeatedly add space on the higher side of the constraint, to bring the vertices of the constraints to the same height.

In the example from Section 3.3 for a longest increasing subsequence we had found 3, 4, 5, 6. The corresponding constraints back in the examples from Section 3.2 were $(3, 3')$, $(4, 4')$, $(5, 5')$ and $(6, 6')$. You can find the adjusted graph drawing in Figure 3.7 and the schematic of the layers after adjustment in Figure 3.8.

If we want to adjust the layouts in such a way that these constraints are horizontal, we have to shift parts of the graphs downwards, changing the layer assignment L to an adjusted one L_A . First, we want to bring 3 and 3' to the same height. For this we can shift the whole graph on the right two layers downwards. The vertex 3' was originally on layer 0, so $L(3') = 0$ and is on layer 2 afterwards, so $L_A(3') = 2$. Next $(4, 4')$ are brought to the same height. We have $L(4) = 3$ and $L(4') = 3$, but the vertex 4' of the right graph was moved two layers downwards by the previous adjustment. So by shifting all vertices of the left graph from layer 3 and below by two layers downwards we get $L_A(4) = 5$ and $L_A(4') = 5$. With the two shifts on each side by two layers, the constraint $(5, 5')$ that was initially horizontal is again horizontal, so we do not have to make any more adjustments for this constraint. Lastly, the constraint $(6, 6')$ should be horizontal. When shifting the remaining vertices of the left graph from layer $L(6) = 5$ and below downwards by one additional layer, we get $L_A(6) = L_A(6') = 8$.

To calculate the adjustment efficiently and define some additional properties, we will use two lists of the same length *positions* and *corrections*, for each graph. Any index i in the lists describes one step of the adjustment. The first list, *positions*, contains the

layer number $positions_i$ from which an adjustment starts. We define that each constraint leads to an adjustment, even if it is a shift by 0 layers. Therefore $positions$ is just the list of all layers in which a selected constraint occurs. The second list, $corrections$ contains the amount of layers $corrections_i$ to insert from the layer $positions_i$ on. For example, $positions_i = 3$ and $corrections_i = 2$ means that layer 3 and below should be shifted downwards by two layers. We need those lists for both graphs, and we distinguish the lists for the right graph as $positions'$ and $corrections'$.

If we have these two lists we can calculate L_A efficiently. For a vertex v of the left graph, find the index i of $L(v)$ or the next lower number in $positions$ using binary search. This means v is between the i th and the $(i + 1)$ th constraints. Then add $corrections_i$, that is, the entry of $corrections$ at index i , to its layer number:

$$L_A(v) = L(v) + corrections_i$$

For a vertex v' of the right graph do the same with $positions'$ and $corrections'$.

In the example from Figure 3.7 the position lists are quickly built with $positions = L(3), L(4), L(5), L(6) = 2, 3, 4, 5$ and $positions' = L'(3), L'(4), L'(5), L'(6) = 0, 3, 4, 6$. The correction list require a little more thought. The first adjustment to bring 3 and 3' to the same height was to shift the whole graph on the right two layers downwards. For this movement store 0 in $corrections$, because the vertices in the left graph are not moved. Additionally, store 2 in $corrections'$, because the vertices in the right graph are shifted by two layers downwards. Next (4, 4') are brought to the same height by shifting all vertices of the left graph from layer 3 down two layers. This means we add 2 to $corrections$ and 0 to $corrections'$. The constraint (5, 5') did not lead to any adjustments, but we defined these to be adjustments by 0 layers. Consequently add 0 to $corrections$ and $corrections'$. Lastly, for the constraint (6, 6') we shift the vertices of the left graph downwards by one additional layer, so we only append 1 to $corrections$ and 0 to $corrections'$. We end up with the lists $corrections = 0, 2, 0, 1$ and $corrections' = 2, 0, 0, 0$.

To show that it is possible to build the lists $corrections$ and $corrections'$ in an efficient manner we use Algorithm 2. It first sorts the list of constraints C in line 3 which can be achieved in $O(|C| \log |C|)$ time. The sorted list is then used as input to LongestIncreasingSubsequence from Algorithm 1 in line 4, which runs in $O(|C| \log |C|)$ time. As seen in the previous sections, the longest increasing subsequence S is a set of non-conflicting constraints. Finally, the loop in line 9 processes each element of S once with constant-time operations, leading to a runtime of $O(|S|)$ for this loop. This leads to a total runtime within $O(|C| \log |C|)$, since S is a subsequence of C and $|S| \leq |C|$. For applying the corrections to a single vertex we get a runtime of $O(\log |S|)$ caused by the binary search to lookup the index in the sorted list $positions$ of length $|S|$.

One problems arises for constraints that cover a big amount of space in the original drawing, when a high number of layer has to be inserted to make them horizontal: The adjustment results in a huge gap in the graph without any vertices in the whole inserted space. This situation does not occur in the example from the previous section, so we use a new example in Figure 3.9. To adjust the graph in Figure 3.9a we have to make the lower constraint connecting the two lowest vertices of the left and right graph horizontal.

Algorithm 2: AdjustLayerAssignment(LayerAssignment L , List C)

input : A relation L of vertices and layers and a list C of constraints, that is, pairs of vertices

output: An tuple of lists ($corrections, corrections'$) of adjustment amounts

```
1 operator  $(v, v') < (w, w') := L(v) < L(w) \vee L(v) = L(w) \wedge L(v') < L(w')$ 
2 operator  $(v, v') <' (w, w') := L(v') < L(w') \vee L(v') = L(w') \wedge L(v) < L(w)$ 
3  $C' = \text{sort}(C, <')$ 
4  $S = \text{LongestIncreasingSubsequence}(C', <)$ 
5  $corrections = \text{list with only element } 0$ 
6  $corrections' = \text{list with only element } 0$ 
7  $c = 0$ 
8  $c' = 0$ 
9 foreach  $(v, v') \in S$  do
10    $offset = L(v) + c - L(v') - c'$ 
11   if  $offset < 0$  then
12      $c = -offset$ 
13      $c' = 0$ 
14   else
15      $c = 0$ 
16      $c' = offset$ 
17    $corrections.append(c)$ 
18    $corrections'.append(c')$ 
19 return ( $corrections, corrections'$ )
```

Our initial algorithm would compare the layers of the vertices and insert two layers above the vertex of the right graph as shown in Figure 3.9b. As you can see this vertex is now separated apart from the other two vertices of the right graph. The adjustment brought the two corresponding vertices of the two graphs closer together to aid the comparison, however, it spread related vertices in the right graph apart, making the drawing less useful overall.

This problem can be mitigated by interpolating between offsets, as shown in Figure 3.9c. Nothing changes for the vertices connected by the constraints. The vertices on the layers between them, however, are distributed evenly across the inserted space. Alternatively this can be described as scaling the part of the drawing between the layers of the constraints.

To calculate the positions for the vertices from the lists $positions$ and $corrections$ we use interpolation. For any vertex v with its layer $L(v)$ between $positions_i$ and $positions_{i+1}$ we use the following equation to calculate L_A :

$$L_A(v) = L(v) + corrections_i + (L(v) - positions_i) \cdot \frac{corrections_{i+1} - corrections_i}{positions_{i+1} - positions_i}$$

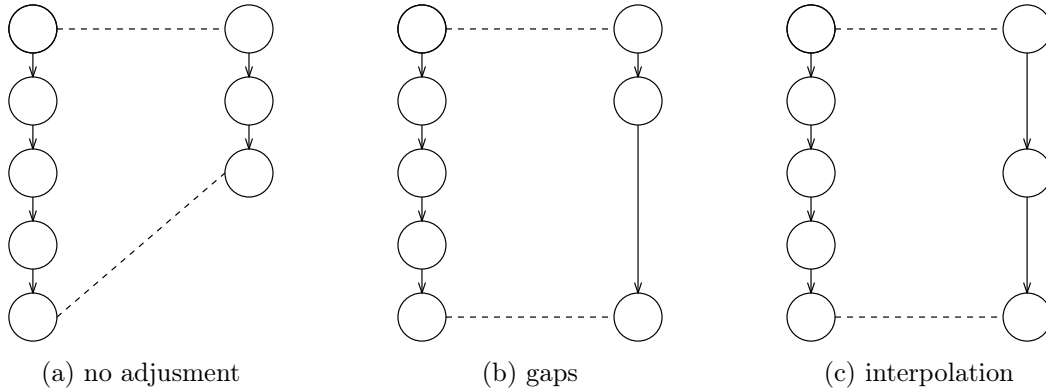


Fig. 3.9: Drawings of two graphs with three different adjustment methods. The graphs are just paths with five and three vertices respectively. The two constraints between the left and the right graph are indicated by the dashed lines.

As one can see by the occurrence of a fraction in this equation the value of L_A may not be an integer but a rational number instead. This indicates that the layer of the vertex should be placed between the positions of the previous integer layers. In this case, L and L_A are analogous to the y-coordinates of the vertices. The denominator $positions_{i+1} - positions_i$ is always greater than zero because constraints in the same layer were excluded when calculating the longest increasing subsequence.

3.6 Adaptive scrolling

Based in the offset lists and adjustment described in the previous section we developed an interactive comparison aid that we call *adaptive scrolling*. The method was inspired by the scrolling behavior of diff-GUIs, that is, applications that compare text in a graphical user interface. Equal lines of text are aligned in a side by side view and differences are highlighted. While scrolling, if one of the texts has extra lines, the other side stays in position until all of the inserted text has passed by. Gleicher et al. [GAW⁺11] showcase a tool called *Vdiff* which apparently exhibits this behavior, however, we were unable to obtain a copy. A similar tool called *Kompare* is readily available¹.

Viewing and comparing large graphs on a computer screen results in a trade-off between recognizable graph elements and readable label texts and the size of the section of the graph that fits on the screen. To fit a large section of the graph drawing into the screen viewport the size of the drawing can be scaled down, known as *zooming out*, which at the same time decreases the size of labels texts and other elements of the drawing. At some point the size becomes too small to read the labels or recognize the graph elements. As a result, with larger graphs only a section of the graph can be shown at a time. The user then navigates through the drawing by moving the viewport. Vertical movement is commonly referred to as *scrolling*.

¹Kompare is available through the KDE project or at <http://www.caffeinated.me.uk/kompare/>.

We developed some approaches to ease scrolling. First, we split the viewport into two halves, each displaying one graph. We added the option to navigate each viewport independently and thus the ability to compare arbitrary parts of each graph even after zooming in. Otherwise the two viewports are moved together. Secondly, we added the option to use adaptive scrolling. It is a hybrid mode between independent and coupled scrolling. The viewport currently scrolled by the user moves in the usual way. The other viewport is moved with increased or decreased scrolling speed to bring the vertices linked to the newly visible vertices of the dragged graph into the other viewport. This interactive process is depicted as a time-series in Figure 3.10. In the first instance the two viewports are centered on the two topmost vertices of both graphs. In this exaggerated example only about two vertices fit into the viewports. As the user scrolls downwards, the graph drawings are moved upwards through the viewports. The left graph is moved faster than the right graph. In the last instance the bottom vertices of both graphs are now centered in the viewport.

It is not trivial to apply adaptive scrolling to graphs with more vertices and more constraints. If there are two conflicting constraints, one graph would have to move backwards to bring the vertex back into the viewport. For a large number of conflicting constraints the scrolling motion would be a mix of forward and backward motions, which makes the process hard to fathom. We can use the methods developed in the previous sections to get a smooth, continuous motion. We use Algorithm 2 to select a maximum set of non-conflicting constraints. By using only these non-conflicting constraints we ensure a continuous forward movement. Also the algorithm builds lists for calculating an adjusted layer assignment L_A from another layer assignment L . The layer assignment adjustment with interpolation can be used to calculate a smooth scrolling motion. If the user is dragging the left graph we look at the point c in the drawing, that is, in the center of the left viewport. The y -coordinate of c defines a rational number indicating its position between the layers of the drawing of the left graph which we write as $L(c)$. If we treat c like a vertex in the left graph, we can obtain an adjusted layer coordinate $L_A(c)$. Next we conceive a vertex c' of the right graph with $L_A(c') = L_A(c)$. If we reverse the adjustment of the right graph we can obtain the corresponding layer coordinate $L(c')$. We set the scrolling position of the right graph so that the $L(c')$ is in the center of the viewport. If the user is dragging the right graph we use the same process the other way round.

As an example, look at the last frame of the time-series in Figure 3.10. Say the user dragged the left graph with the five vertices to obtain this situation. The center c of the left viewport is just the vertex on the bottom, which is on layer 5 in the drawing, thus $L(c) = 5$. To obtain $L_A(c)$ look at the adjusted drawing in Figure 3.9c. The vertex is on layer 5 in this drawing, too, so $L_A(c) = 5$. Next we conceive c' in the right graph with $L_A(c') = 5$. As depicted in Figure 3.9c, there is actually a vertex in layer 5 in the adjusted drawing of the right graph, the third vertex from the top. This vertex was on layer 3 before the adjustment. This means we have to move layer 3 of the right graph into the center of the viewport. This is indeed what happens in Figure 3.10.

In conclusion, adaptive scrolling is easily realized using the same calculations as in the adjustment with interpolation. It does not change the graph drawing itself, so it may

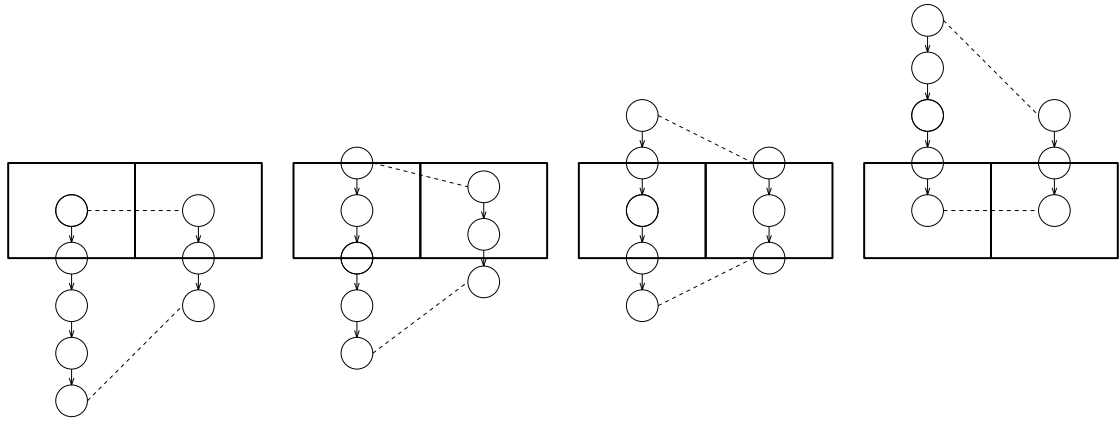


Fig. 3.10: Time-series of navigating through a drawing of the graphs from Figure 3.9a using adaptive scrolling. The bold rectangles depict the viewports. Scrolling down is depicted by moving the graphs upwards through the viewport.

be used for complex drawings where inserting space between layers is not feasible. It is an interactive method, so it cannot be used in cases where scrolling and dragging of the drawings through a viewport is not applicable, for example, for a paper printout of the graphs or when the graphs fit into the viewport entirely.

4 Evaluation

So far, Chapter 2 described the relationship between graphs and business processes and Chapter 3 introduced methods for assisting users in visual process comparisons. This chapter is about the evaluation of these methods and if they prove themselves in practice. Initially, Section 4.1 describes a tool previously developed by Andrews et al. [AWW09] and the underlying algorithm. Their tool, like ours, was developed to compare business processes and we will look at differences and common features. Next, Section 4.2 describes the comparison tool prototype that we developed to make the algorithms applicable to real-world problems. Lastly, we did a user study where we asked the participants various questions to assert the usefulness and other properties of our approach.

4.1 Placing similar vertices on the same positions

We were looking for other tools that employ aids for visually comparing business processes. In 2009, Andrews et al. [AWW09] presented a tool for comparing and merging business processes called *Semantic Graph Visualiser* (SGV). The SGV works on two similar input graphs G_1 and G_2 and a list of node similarities. The tool presents the two processes side by side and a third view in between them shows a merged graph G_m . The user can then manage node similarities and adjust the layouts. A toolbar displays the similarity matrix and a small overview of the merged graph for navigation, replacing the feature of zooming in and out. The SGV works on general graphs and has no EPC specific features, like different shapes for functions, events and connectors.

To make the graphs visually comparable and create a merged graph, a multi-step process is used. First, the graphs and the list of similarities are loaded. Since we have lists of constraints they are converted into similarities by keeping the pairs and assuming a similarity value of 1. The similarity list is then converted into a similarity matrix by setting the corresponding entries. If a pair of vertices does not occur in the list, their similarity is set to 0.

The next step is to chose matching vertices. A vertex can be similar to an arbitrary number of other vertices from the other graph, but for the graph merging it can only be assigned to one other vertex or to no other vertex. This assignment problem can be solved in $O(n^3)$ time using the *hungarian algorithm*, where n is the number of vertices.

Next, a merged graph G_m is constructed from the graphs G_1 and G_2 . The merged graph contains all vertices and edges from G_1 . Additionally, it contains all vertices from G_2 which are not matched with another vertex from G_1 . The edges from G_2 cannot be directly added, because the vertices with matches do not occur in G_m . Whenever there is an edge in G_2 to (from) a vertex with a matching vertex in G_1 an edge to (from) this matching vertex is inserted instead.

As a last step a layout for G_m is calculated and then used to layout G_1 and G_2 . The vertices of G_1 and the vertices of G_2 without a match get same positions as in the layout of G_m . The vertices of G_2 with a match get the position of their corresponding vertex from G_1 . The SGV allows users to inspect and adjust the results of each of the intermediate steps.

Figure 4.1 shows an example for a comparison created in this way. The two graphs G_1 and G_2 are the same graphs already used as examples in Chapter 3, but this time with the layout derived from the merged graph G_m . All vertices with numbers in G_1 were matched with their counterparts in G_2 and are at the same positions in the drawings of both graphs. The unlabeled vertices remain unmatched and there are gaps in their place in the respective other graph.

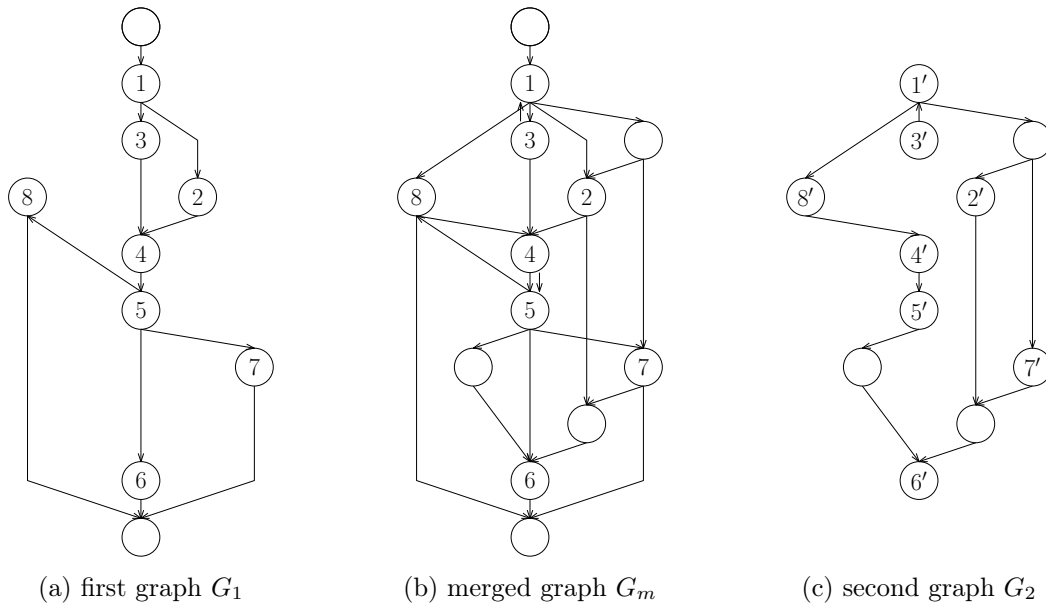


Fig. 4.1: Drawings of two graphs 4.1a and 4.1c with the layout of their merged graph 4.1b. The graphs are the same like in Figure 3.1.

There are some fundamental differences to the algorithm we presented previously to bring vertices in a constraint pair to the same height. For the layout of a merged graph we must run a layout algorithm. This can be an advantage, since almost any layout algorithm can be used, whereas the same height adjustment can only meaningfully be applied to layered graph drawings or other drawings that can easily be spread apart in height. But it can also be a disadvantage. A time-expensive layout algorithm might require a long time to calculate the layout for the merged graph, while the same height adjustment has tight theoretical runtime bounds. Also, if an automatic layout is not sufficient, manual adjustments to the merged layout have to be made. The same height adjustment can work on manually created layouts. In case of the scrolling adjustment, the original layouts are not changed at all. Both offer no additional support when the task is to merge two graphs.

When it comes to specifying differences and similarities between graphs, the SGV is a bit more flexible. It uses a list of similarities with weights. Our methods currently only uses binary constraints, that is, a constraint either connects two vertices or not. However, both tools have to select a set of similarities. The SGV uses the hungarian algorithm with an $O(n^3)$ runtime bound, while our methods use an algorithm for finding a longest increasing subsequence with an $O(m \log m)$ time bound, where m is the number of constraints.

4.2 Development of an interactive comparison tool

To test the algorithms described in Chapter 3 and in the last section we built a custom prototype tool. Figure 4.2 shows a screenshot of the user interface. This prototype provides a set of tools to compare business processes or – more specifically – EPCs. It was realized in the programming language *Java*. Using the Java technology, the tool runs on a virtual machine which is available on many platforms. Also, there are some libraries available for Java solving various tasks in the fields of graph drawing and business processes, some of which were leveraged for development of the tool.

The tool was designed to allow displaying EPCs side by side. It also contains rudimentary functionality to manage constraints on the graphs. The main focus was on the graph layouts and the interactive comparison. To keep things simple, it consists only of a toolbar and the graph views. The buttons of the toolbar were ordered from left to right in the order of the workflow when comparing EPCs. The first button can be used to load EPC files. The tool can load an arbitrary number of EPCs only limited by available screen space and memory. The files are expected in the XML-based format of the *bflow* Toolbox*¹ which is a tool for modeling and simulating EPCs and was introduced by Böhme et al. [BHK⁺10]. The second button is used to load a repository of constraints. The format for the constraints was developed specifically for the needs of the prototype. It is XML-based, too, and provides a list of pairs of identifiers for the EPCs' elements. In the center there is a button for adding new constraints. The user can select EPC elements by clicking and they will be linked with a new constraint when clicking this button.

Next in the toolbar, a drop-down menu is provided. It is used to switch between the various adjustments, namely the interpolated height adjustment from Section 3.5, the adaptive scrolling as described in Section 3.6 and the adjustment based on merging graphs, which was described in the previous Section 4.1. There is also an option to not use any adjustments. Lastly, there are two buttons to save the constraint repository to a file and to close all EPCs. These are used when starting a new comparison. Unlike to the SGV, the tool does not have an additional view for a merged graph, but only displays the two graph with the layout of the merged graph applied to them. All in all, the buttons of the toolbar provide the essential functionality to control the comparison.

The second and more prominent part of the tool is the graph view below the toolbar. It contains the graph drawings for the opened EPCs. It is vertically split into multiple

¹bflow* Toolbox is available at <http://bflow.org/>

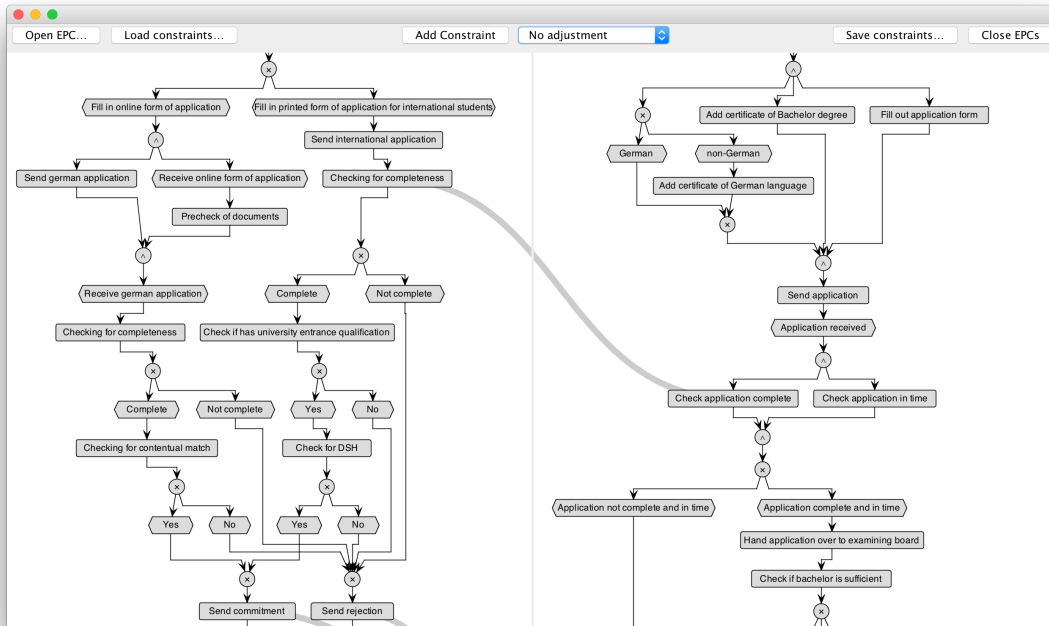


Fig. 4.2: Screenshot of the interactive comparison tool.

panes, one for each EPC. The graphs in these panes can be dragged and the mouse wheel – or a substitute input method for notebook computers – can be used to zoom in and out. In the default mode all graphs move together. Holding down a modifier key the graphs of each individual pane can be moved independently. The panes cannot be zoomed independently, as this would introduce complexities in the movement behavior.

Constraints are depicted as thick semi-transparent lines. Their thickness and transparency distinguishes them from regular edges. Opaque lines caused problems when many constraints ran in almost parallel paths. To further differentiate them from other edges we used Bézier curves instead of straight lines. The two intermediate Bézier control points are horizontally centered between the vertices and distributed in their heights to the heights of the two vertices. After noticing completely horizontal constraints overlapping other vertices, we used a slightly curved line in those cases.

The graph panes were realized using *JUNG*², the *Java Universal Network/Graph Framework*. This library provides data structures for graphs and interactive visualizations. It was introduced by O'Madadhain et al. [OFS⁺05]. The graph layouts were calculated using algorithms of the *KLay*³ library, which is part of the *Kiel Integrated Environment for Layout Eclipse RichClient* or short *KIELER*. It provides Sugiyama-style graph drawing including heuristics as described by Schulze et al. [SSH14].

²JUNG is available at <https://github.com/jrtom/jung>

³KLay is available at <https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered>

The tool was tested on EPCs provided to us by the Komplex-e project. The EPCs are reference processes, modeling either Plan-to-Produce or Forecast-to-Fulfill procedures. They were derived from four business-management software applications⁴. This means there are four EPCs in each of the two categories. Each set of four EPCs can be assembled into six pairs, yielding twelve process pairs in total. The EPCs have between 61 and 132 elements and 60 to 143 connections. The pairs have between 14 and 48 constraints, 27.83 on average. The speed of the tool on these process pairs was good enough, such that no significant delay was noticeable for calculating the layouts or the adjustments.

When inspecting the changes in size caused by the adjustments for the test processes, a pattern emerges. In Figure 4.3 each plot contains one arrow for each of the twelve pairs of processes in our test set. The base of the arrow is at the size of the original drawings. The tip of the arrow is at the size of the adjusted drawings. The size is calculated by adding the widths of both drawings and taking the maximum of both heights. This resembles placing the two drawings side by side on paper. The plot in Subfigure 4.3a shows that the merged graph adjustment increases the width, sometimes by large amounts, while the height stays almost the same or even goes down. In our test set the width increased by 38–258 % and on average by 128 %. The large increases in width are caused by the small amounts of matched vertices. When two rather linear processes are merged and the merged graph is no longer as linear as the source processes, the Sugiyama algorithm can place vertices next to each other, leading to a reduction in height. The largest reduction was by 11 % of the original height, opposed to a maximal increase of 48 %. The average height increase of 6 % is remarkably close to no increase at all.

The plot in Subfigure 4.3b shows the increase in size when using the same heights adjustment. Because only vertical space is added the width does never change and all arrows point straight upwards. The increase of height fluctuated between 3 % and 46 %, averaging at 22 %. A small increase of 3 % can occur when a smaller business process is scaled up beside a larger one.

4.3 Design of the user study

After the theoretical foundation for various comparison aids was laid out and implemented in a usable way, we conducted a user study to learn how our tool would work out in practice. The user study proceeded in two stages. The first stage was to conceive, test and improve the survey design and the tool. In the second stage a number of individuals was asked to participate in the revised survey.

For the first stage, we prepared an instructional text to explain the basic functionality of the tool and a total of 79 questions about various aspects of the tool. Some questions were available in different wordings. The questions were aiming to assert usability of the split graph view and the four different methods of adjustment implemented. Participants were instructed to test and compare each of the methods before starting the survey, to make the scores comparable regardless of the order of the questions for the adjustments.

⁴The applications are: Microsoft Dynamics NAV, SAP Business ByDesign, Sage Office Line and godesys.

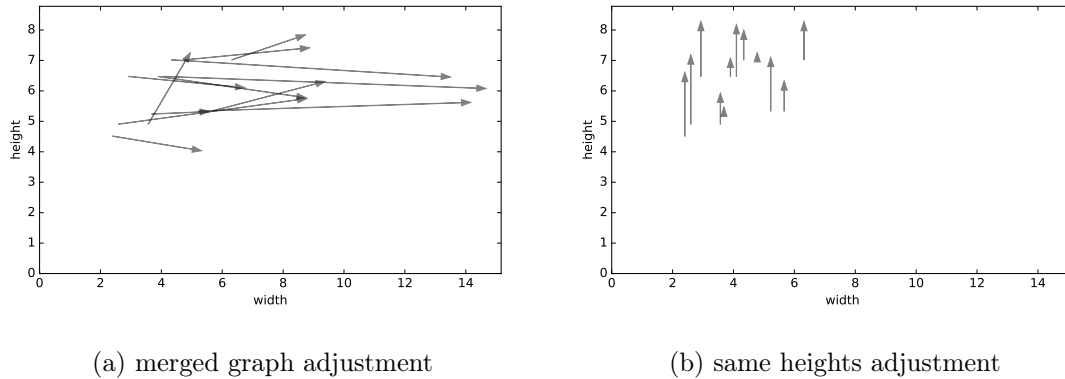


Fig. 4.3: Size increase when applying different adjustments to example process flowcharts. The arrow points from the original size to the adjusted size.

Otherwise assigning a high or low score in the beginning renders assigning an even higher or lower score later impossible.

After drafting a wide range of questions, we conducted a supervised participation with two test participants. They read the instructions, used the tool as described in the instructions and filled out the questionnaire. They were instructed to think aloud throughout the whole process. Using this feedback we identified three main weaknesses with the study and proceeded to remedy or fix them.

The first weakness becoming apparent were the complex instructions. To simplify the instructions and to reduce barriers for participation we modified the tool used for testing and added a mechanism that pre-loads two example processes into the comparison view. This had the side effect that we could now also ask participants which processes they used while testing and we were hoping to collect additional data on how the choice of the process influenced the overall experience.

The second weakness indicated by the participants was that the processes were too complex to grasp in a reasonable amount of time. We mitigated this issue by clarifying the detail of required comparison in the instructions. We also added a simpler example. To ensure that complex processes are still covered, too, we selected three different pairs of processes. Two pairs of processes were selected from the Forecast-to-Fulfill dataset from the Komplex-e project, henceforth labeled as *Example A* and *Example B*. Another pair of processes was derived from the data set of the 2015 process model matching contest of Antunes et al. [ABB⁺15]. This contest uses three different kinds of processes. The first dataset *Asset Management* contains of 72 EPC models from the SAP Reference Model Collection and is similar to the dataset we used from the Komplex-e project. The second dataset *Birth Registration* contains models representing birth registration processes in various countries. These models are only available as petri-nets, which cannot be loaded into our tool. The third dataset *University Admission* contains 9 models representing the application procedure for Master students of nine German universities in BPMN format. All datasets include a gold standard, a description of ideal matchings fostered

manually. Since the processes of examples A and B were considered very complex by the test participants, a simple processes different from examples A and B was chosen. Two processes from the University Admission dataset were picked and converted into the EPC format used by the tool. The pair was added as *Example C* to the user study. The processes for examples A and B consist of 79, 90 and 132 elements and have 14 and 34 constraints between them. The processes for Example C consist of 49 and 40 elements and there are 7 constraints from the corresponding gold standard. The participants were randomly given one of the examples.

The third weakness was that some questions were identified as vague when the participants asked for clarification. We removed some questions that lead to misunderstandings and reworded some others to clarify. Based on additional feedback of the test participants, we checked if the questions were interpreted the way we planned. We ended up with a total of 42 revised questions. Three questions are about previous knowledge, five questions about the evaluation itself and eight questions about properties of the tool in general. For each of the three adjustments – not including no adjustment at all – there were six questions about properties of the adjustments. Another eight questions asked which of the four adjustments – now including no adjustment at all – fits best to a given property. This last group of questions was used to check the plausibility of the other answers. If the participants were asked for a score, they had to answer with a number between 1 and 5 with 1 standing for “I do not agree at all” and 5 standing for “I fully agree”. An odd number was chosen to allow an undecided answer.

These questions and instructions were then composed into an online questionnaire and sent to the members of a chair of computer science and the members of an institute of business informatics. The instructions included steps to obtain and launch the tool and basic information about the purpose of the tool. Even though all members had the same chances to participate, we can assume an influence of self-selection, since we did not check which individuals invited actually submitted answers and individuals only responded when motivated. The selection of possible participants also assumes that academics can judge a tool intended for practitioners. We were also interested to see if individuals without sophisticated prior knowledge in the fields of computer science, informatics and economics would be able to use the tool, so we had two other students from unrelated fields participate. To reduce problematic submissions, the participants were given the option to mark their submission as unfaithful or skip the questions if the tool does not work at all for technical reasons.

4.4 Results of the user study

After designing and revising the survey in the first stage, we collected answers from participants in the second stage. All in all, the study had 13 participants. Eight participants were from the field of computer science, three of which had already been studying for over 2 years, another three had completed a degree and another two had completed their doctoral studies. All but one of the computer science educated participants stated that they had already concerned themselves with graph theory. This might indicate that only

those interested in graph theory were motivated to respond, confirming suspicions about self-selection, on the other hand it is one of the most common fields of study.

Two participants had graduated in the field of business informatics and one was currently a student of economics. They all said they had experience with business processes. Two of them had studied them in a scientific context, one of which also had concerned himself with graph theory. Additionally, two participants were not related to computer science nor economics nor business, one of which had graduated in an unrelated field.

Participants were instructed to rate their approval to various statements on a scale from 1 to 5. The average score of n score ratings x_1, \dots, x_n is calculated with

$$\bar{x} = \frac{1}{n} \sum_{i=0}^n x_i$$

which is also called *arithmetic mean*. The *corrected sample standard deviation* was chosen because the actual mean score in the statistical population is not known. It is calculated as

$$s = \sqrt{\frac{1}{n-1} \sum_{i=0}^n (x_i - \bar{x})^2}$$

and s^2 is the related *variance*. For example, the scores 1, 2, 3, 4, 5 imply $\bar{x} = 3$, $s^2 = 10/4 = 2.5$ and $s = \sqrt{2.5} \approx 1.58$. Assuming all scores are equally likely, the expected value $E(s)$ of the standard deviation s is 1.41. An overview of all score averages and corrected sample standard deviations can be found in Table 4.1. The questions with the lowest standard deviations are about the complexity of the tool and about the self-assessed faithfulness of the answers. Participants generally claimed to answer faithfully with a mean score of $\bar{x} = 4.62$ on a scale from 1 to 5 and perceived the complexity of the tool as rather low with a mean score of $\bar{x} = 1.69$. The navigation through the graph layouts with coupled ($\bar{x} = 4.23$) and independent ($\bar{x} = 4.23$) scrolling was generally considered helpful, indicating that both modes are necessary. Participants were undecided about the overall helpfulness of the tool with a mean score of $\bar{x} = 3.46$ and a standard deviation of $s = 1.33$. In hindsight, the lack of a clear object of comparison was an error in the concept of this question. Even though the scores for Example C seem in general more unfavorable, they may be biased, because the example was only assigned to two participants, both from the computer science subgroup. The scores of participants that looked at Example A are in general more favorable than those of participants with Example B. Since Example B is more complex – 34 constraints versus 14 in Example A – we can conclude that the processes had an influence on the ratings of the tool.

A set of scores were collected for each of the three evaluated adjustments. When asked about the desirable properties – helpfulness, predictability or intuitiveness – participants tended to give on average higher scores to the same heights adjustment that made selected constraints horizontal by spreading certain parts of the layout. The adaptive scrolling adjustment that only changed the scrolling with respect to some constraints ranked second. The merged layout adjustment which layouts both processes based on

Tab. 4.1: Overview of scores allotted by survey participants, mean scores \bar{x} and corrected sample standard deviations s . Additional columns for subgroups of the participants, those with economics background and those using one of the three provided examples. The numbers in parentheses indicates the number of participants in this subgroups. The second part lists properties for each of the adjustments, with the key SH = same heights, ML = merged layout, AS = adaptive scrolling for brevity. The values for the property desirability were derived from the other questions.

Property	Adj.	All (13)		Econ. (3)		Ex. A (5)		Ex. B (6)		Ex. C (2)	
		\bar{x}	s	\bar{x}	s	\bar{x}	s	\bar{x}	s	\bar{x}	s
Helpfulness		3.46	1.33	3.33	2.08	4.40	0.55	2.83	0.98	3.00	2.83
Clarity		3.54	1.20	4.00	0.00	4.00	0.71	3.50	1.22	2.50	2.12
Unintelligible		2.15	1.21	2.67	2.08	1.60	0.55	2.67	1.51	2.00	1.41
Complexity		1.69	0.85	1.67	0.58	1.40	0.55	2.00	1.10	1.50	0.71
Process Layout		3.38	1.04	4.00	0.00	3.80	0.84	3.33	0.82	2.50	2.12
Helpfulness of moving together		4.23	1.09	4.33	1.15	5.00	0.00	3.83	0.98	3.50	2.12
Helpfulness of independently moving		4.00	1.08	3.67	1.53	4.60	0.55	3.67	1.03	3.50	2.12
Confidence in own answers		3.31	1.03	4.33	0.58	3.60	1.14	3.17	0.98	3.00	1.41
Answer faithfulness		4.62	0.65	4.67	0.58	4.60	0.89	4.67	0.52	4.50	0.71
Helpfulness	SH	3.92	1.12	3.67	1.53	4.40	0.89	3.67	1.03	3.50	2.12
	ML	2.92	1.38	4.33	0.58	3.40	1.82	2.83	1.17	2.00	0.00
	AS	3.30	1.60	2.67	1.00	4.00	1.22	2.33	1.63	4.50	0.71
Complexity	SH	1.46	0.78	2.33	1.00	1.40	0.55	1.67	1.03	1.00	0.00
	ML	2.61	1.33	2.00	1.53	2.60	1.52	2.33	1.03	3.50	2.12
	AS	2.38	1.33	3.00	0.58	2.20	1.30	3.00	1.26	1.00	0.00
Predictability	SH	4.00	1.00	4.00	1.15	4.20	0.84	4.00	0.89	3.50	2.12
	ML	3.07	1.19	3.67	1.00	3.00	1.22	3.17	1.33	3.00	1.41
	AS	4.00	1.15	3.00	0.58	3.80	1.30	4.00	1.26	4.50	0.70
Helpfulness in all circumstances	SH	3.00	1.15	3.00	0.00	3.40	0.89	2.50	1.05	3.50	2.12
	ML	2.92	1.04	3.00	0.58	2.40	0.89	3.17	0.75	3.50	2.12
	AS	3.15	1.52	3.00	0.58	3.80	1.30	2.50	1.52	3.50	2.12
Negative emotions	SH	2.08	1.26	2.33	2.08	2.00	1.22	2.17	1.47	2.00	1.42
	ML	2.92	1.38	2.33	1.73	3.00	1.41	2.83	1.60	3.00	1.42
	AS	2.23	1.36	2.33	1.73	2.20	1.30	2.67	1.51	1.00	0.00
Intuitiveness	SH	4.38	0.65	3.67	1.73	4.60	0.55	4.17	0.75	4.50	0.71
	ML	3.62	1.04	3.67	1.53	3.40	0.89	3.83	0.98	3.50	2.12
	AS	3.85	0.99	2.67	1.15	3.80	1.10	3.67	1.03	4.50	0.71
Desirability	SH	4.15	0.88	3.73	0.82	4.36	0.81	4.00	0.81	4.10	0.92
	ML	3.22	0.72	3.87	0.41	3.24	0.49	3.33	0.56	2.80	0.84
	AS	3.71	0.74	3.00	0.68	3.84	0.37	3.27	0.64	4.70	0.39

a merged graph ranked third. When asked about the undesirable properties – added complexity or the presence of negative emotions – the ranks are inverted. This suggests a trend, however, the differences are smaller than the standard deviations, with only two exceptions. Participants rated the perceived increase in complexity for the same heights adjustment at a low score of $\bar{x} = 1.46$ with an unusually small standard deviation of $s = 0.78$. The only other value with a smaller standard deviation of $s = 0.65$ is the rather high score of $\bar{x} = 4.38$ for the intuitiveness of the same heights adjustment. A total of 69% of the participants selects scores of 4 or 5 for the helpfulness of the same height adjustment, as opposed to only 38% for the merged layout adjustment and 54% for the adaptive scrolling. When asked which of the adjustments is the most reliable, 31% opted for the same heights. In summary, the height adjustment felt most natural and unintrusive to the participants.

To embrace the concept of desirable and undesirable properties we derived a new property *desirability*. For each participant and each adjustment look at the desirable properties scores. Add the undesirable properties subtracted from 6. This will inverse these scores. The average of these values together is a measure for the overall quality of the adjustment perceived by this participant. The standard deviation of these values indicates how coherent the participant answered the questions or how correlated these properties are. Since the standard deviations were generally low we concluded that the properties are indeed correlated. One participant had outstandingly high standard deviations for each of the derived scores. We traced the origin back to situations like them having bad feelings about an otherwise helpful adjustment. We first suspected an error in the calculations or a fraudulent submission. The calculations were double checked and the other answers were in check, too, so we dismissed this theory. The average values found in Table 4.1 are the arithmetic means of the participants' averages. The standard deviations are the square roots of the arithmetic means of the participants' variances. Thus, they reflect the total average and standard deviation across all desirability properties and participants.

When asked to name the most helpful or the most functional adjustment, most participants – 5 of 13 – picked the merged graph adjustment. The three participants with economics background – including business informatics – ranked the three adjustments in the opposite way, putting the merged graph adjustment in front. Especially the helpfulness they scored highest for the merged layout with $\bar{x} = 4.33$ and a low standard deviation of $s = 0.58$. One can only speculate why, maybe they are more familiar with the concept of merging business processes.

Participants were offered to write textual responses about their experience with the tool and their experience with the evaluation itself. When asked about the evaluation, one user answered that they would have wanted more processes for testing. It was considered to add more processes, however it would be hard to tell which processes the participants looked at and which processes influenced their final scores most. The responses about the experience with the tool provide additional insights.

Three participants encountered some minor bugs or inconveniences, like the view not updating immediately after adding a constraint. These bugs were subsequently fixed and are thought to not have influenced the rating of the adjustments but might have

influenced the scores for the overall usability of the tool negatively.

Three participants explicitly noted that they did not understand the merging adjustment or did not consider it intuitive. This is probably due to the processes having very few nodes in common. One of the participants stated that the layouts became too wide for the viewport. Andrews et al. [AWW09] note that “if [the graphs] are too dissimilar, then the merge graph is meaningless”. They do not give a fixed threshold, but from the results of the evaluation we can assume that the merged graph was at least in part meaningless.

One participant stated that they would like a different representation of the constraints when the vertices are on the same height, because of excessive overlap. The tool already draws the constraints as arcs in this case, but apparently this attempt to untangle constraint lines is futile and a better solution is needed. This is a disadvantage of bringing the vertices to the same height which we could not solve.

One participant noted that adaptive scrolling is only useful if the viewport shows just a small part of the drawing, confirming our similar conjecture. Two participants expressed their wish for more color in the tool, one even suggested colorful icons for the buttons in the toolbar. Color would definitely add value to the tool, but its use should be considered carefully and its artistic nature was out of scope for now. One participant acknowledged that they had a short phase of disorientation in the beginning. One participant found a typo in one of the provided processes. The fact that only one single participant found this typo might indicate that participants overall did not look into the specifics of the processes but we think this does not diminish the value of the evaluation.

Because of the small sample sizes, it is hard to derive statistically significant results. However, strong effects would stand out, so we can conclude that there were no severe issues with any of the adjustments or the tool itself. On the other hand, no single adjustment was considered clearly superior. We would therefore suggest letting the user choose an adjustment and augmenting comparison tools with additional aids.

5 Conclusion and Future Work

There are many approaches to help visually comparing business processes and graphs in general. We focused on three methods based on adjusting the layout of the graphs. Two of the methods are based on an algorithm which uses a maximum independent set of a permutation graph to select pairs of corresponding vertices which can be brought to the same height by inserting new layers into one of the drawings. One of the methods brings them to the same height by adjusting an existing drawing and the other one adapts the vertical scrolling motion to show related parts on the screen. A third method layouts a merged graph of the two graphs compared and propagates the result back to the graphs.

All three algorithms were implemented in a prototype tool which allows displaying and comparing of business processes in EPC format. The implementation of the algorithms was rather straightforward compared to the integration and customizing of the various libraries for loading, displaying and interacting with business processes.

The user study concluded that the height adjustment felt most intuitive and least obtrusive to the participants. The layout based on a merged graph was considered functional and useful by some and confusing by others, probably founded on the fact that not many elements could be matched in our test EPCs. The adaptive scrolling method drew mixed reactions, but was considered useful by some. Since the answers did not allow a clear conclusion, maybe the comparison aids have to be evaluated in depth by experts and in more real-world scenarios. It is probably best to choose an adjustment best suited to exact use-case and processes. The layouts using the merged graph seemed superior for graphs with many matching nodes or if the aim of the comparison is to create a merged graph.

We showed how transferring business processes into graphs enables the use of known layout algorithms to draw neat diagrams. In the future, it would be interesting to compare the influence of the layout adjustments to some more artistic visual aids, like using colors to highlight similar elements or changes. Another possible approach is the extension of the longest increasing subsequence algorithm to the weighted problem. Then, each constraint has a weight value and the task is to find a set of non-conflicting constraints of maximum weight. This way, important or low-effort constraints could be preferred when selecting the constraints to cater the adjustment to. This could reduce edge cases where single constraints induce great changes.

Changes to the way the constraints are visualized could improve the readability of the diagrams. Currently there are many crossing lines in areas with many constraints. Also, the case of $n : m$ matchings, where a whole group of elements is related to another group, is currently uninvestigated for all three methods. A solution may be merging them into new vertices. For these matchings a new way of visualizing the constraints could be introduced, since a simple line connecting two elements is only reasonable for

the case of 1 : 1 matchings. Figure 5.1 shows how $n : m$ matchings are visualized in some publications. The related elements of both processes are wrapped in colored shapes which make a confusion with edges unlikely. In case of a simple process this is trivial and was performed manually in this instance, but for more elaborate processes with complex relationships it becomes a lot harder to route the boundaries of the shapes around unrelated elements and keep them free of overlays.

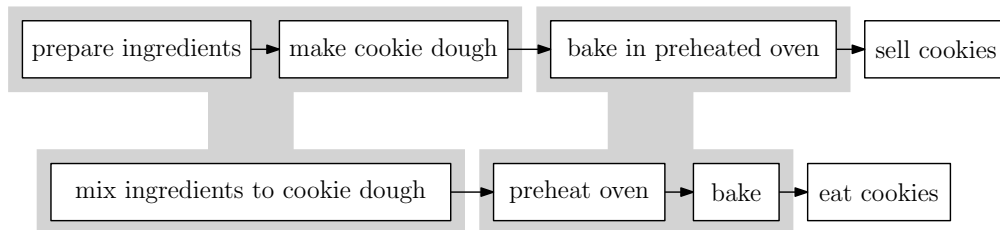


Fig. 5.1: Example for two processes each about making cookies. Here, the matchings include more than two elements and are displayed using simple rectilinear gray polygons.

Another open problem is applying the methods in those areas where a simple graph abstraction is not sufficient to describe the models. One case would be processes with elements grouped into hierarchically nested subprocesses. They can be used to hide complex behaviors and condense them into one element. The corresponding abstraction is compound graphs.

When better automatic comparisons for graphs using semantic analysis and artificial intelligence become available, the need for manual comparison aids will eventually surge. In the meantime, our results might help making manual comparisons a little easier despite their recognized limitations.

Bibliography

- [Aal99] Wil M.P. van der Aalst: Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650, 1999.
- [ABB⁺15] Goncalo Antunes, Marzieh Bakhshandeh, Jose Borbinha, Joao Cardoso, Sharam Dadashnia, Chiara Di Francescomarino, Mauro Dragoni, Peter Fetteke, Avigdor Gal, Chiara Ghidini, *et al.*: The process model matching contest 2015. *GI-Edition/Proceedings: Lecture notes in informatics*, 248:127–155, 2015.
- [AD99] David Aldous and Persi Diaconis: Longest increasing subsequences: From patience sorting to the baik-deift-johansson theorem. *Bulletin (New Series) of the American Mathematical Society*, 36:413–432, 1999.
- [AWW09] K. Andrews, M. Wohlfahrt, and G. Wurzinger: Visual graph comparison. In *13th International Conference Information Visualisation*, pages 62–67, July 2009.
- [BHK⁺10] Christian Böhme, Jörg Hartmann, Heiko Kern, Stefan Kühne, Ralf Laue, Markus Nüttgens, Frank J. Rump, and Arian Storch: bflow* toolbox – an open-source modeling tool. In *Business Process Management Demonstration Track*, number 615 in *CEUR Workshop Proceedings*, pages 46–51, Hoboken, USA, 2010. <http://ceur-ws.org/Vol-615/paper15.pdf>.
- [DDV⁺11] Remco Dijkman, Marlon Dumas, Boudewijn Van Dongen, Reina Krik, and Jan Mendling: Similarity of business process models: Metrics and evaluation. *Information Systems*, 36(2):498–516, 2011.
- [EPL72] S. Even, A. Pnueli, and A. Lempel: Permutation graphs and transitive graphs. *Journal of the ACM*, 19(3):400–410, July 1972.
- [GAW⁺11] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D. Hansen, and Jonathan C. Roberts: Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, 2011.
- [Gol04] Martin Charles Golumbic: *Algorithmic graph theory and perfect graphs*, volume 57 of *Annals of Discrete Mathematics*, chapter 7, pages 157–170. North-Holland, Amsterdam, The Netherlands, 2nd edition, 2004.
- [HN01] Robert Hansen and Gustaf Neumann: *Wirtschaftsinformatik I*. Lucius und Lucius, Stuttgart, 8th edition, 2001.

- [HW08] Danny Holten and Jarke J. van Wijk: Visual comparison of hierarchically organized data. *Computer Graphics Forum*, 27(3):759–766, 2008.
- [Kim90] Haklin Kim: Finding a maximum independent set in a permutation graph. *Information Processing Letters*, 36(1):19–23, 1990.
- [MD06] Aldo de Moor and Harry Delugach: Software process validation: comparing process and practice models. In *Proceedings of the 11th International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD)*, pages 533–540, Namur, Belgium, 2006. Namur University Press.
- [OFS⁺05] Joshua O’Madadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan Biao Boey: Analysis and visualization of network data using jung. *Journal of Statistical Software*, 10(2):1–35, 2005.
- [Sch03] Falk Schreiber: Visual comparison of metabolic pathways. *Journal of Visual Languages & Computing*, 14(4):327–340, 2003.
- [SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden: Drawing layered graphs with port constraints. *Journal of Visual Languages & Computing*, 25(2):89–106, 2014.
- [STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda: Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 19. April 2016

.....
Bernhard Häussner