INSTITUT FÜR INFORMATIK

Preserving Order during Crossing Minimization in Sugiyama Layouts

Sören Domrös and Reinhard von Hanxleden

Bericht Nr. 2103
Nov 2021
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Preserving Order during Crossing Minimization in Sugiyama Layouts

Sören Domrös and Reinhard von Hanxleden

Bericht Nr. 2103
Nov 2021
ISSN 2192-6247

E-mail: {sdo, rvh}@informatik.uni-kiel.de

An abridged version of this report will appear in the *Proceedings of the 13th International Converence on Information Visualization Theory and Applications (IVAPP 2022)*, which is part of VISIGRAPP, the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, Online Streaming, 6 — 8 February, 2022

Contents

List of Figures

Abstract

The Sugiyama algorithm, also known as the layered algorithm or hierarchical algorithm, is an established algorithm to produce crossing-minimal drawings of graphs. It does not, however, consider an initial order of the vertices and edges. We show how ordering real vertices, dummy vertices, and edge ports before crossing minimization may preserve the initial order given by the graph without compromising, on average, the quality of the drawing regarding edge crossings. Even for solutions in which the initial graph order produces more crossings than necessary or the vertex and edge order is conflicting, the proposed approach can produce better crossing-minimal drawings than the traditional approach.

1 INTRODUCTION

Edge crossings are the most important syntactic aesthetic criterion for node-link diagrams (Pur97). However, the desire for few edge crossings should not hinder us in synthesizing, automatically and in real-time, a diagram that abides the Nothing is Obviously Non-Optimal (NONO) principle (KDMW16). The SCChart (vHDM$^+$14) in Fig. 1.1b has no edge crossings, but the drawing is obviously non-optimal when considering the transition order specified in the textual source in Fig. 1.1a. Specifically, the order of edges in the drawing is not consistent with the order of the corresponding transitions in the graph.

In general, graph drawing algorithms consider graphs to consist of unordered sets of vertices and edges. This is also the case for the Sugiyama algorithm (STT81), also known as the layered or hierarchical algorithm, that is used to produce the drawing in Fig. 1.1b. However, in practice we often want to consider some ordering, e. g. the textual order defined in some input file, e. g. in a textual SCChart depicted in Fig. 1.1a or in a .dot file (EGK$^+$02), see Fig. 3.1a. This paper presents an approach to produce drawings where the edges and vertices are ordered in the graph model whenever that is possible without increasing the number of edge crossings, see Fig. 1.1c.

Preserving the textual order in the diagram is part of secondary notation (Pet95) since the visual complies with the semantics. Furthermore, since vertices and edges are ordered as in the graph model, we expect that the layout stability, and with it the preservation of the mental map (ELMS91; MELS95), is improved, since small changes in the graph model do not cause large changes in the drawing.

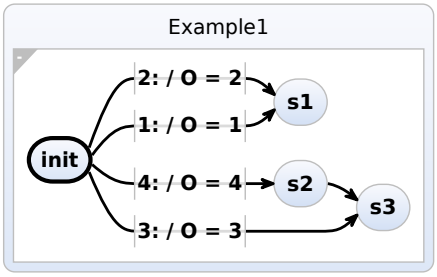
1.1 Contribution & Outline

This paper presents an approach to introduce the concept of graph order to the Sugiyama algorithm. Specifically, the contributions are the following:

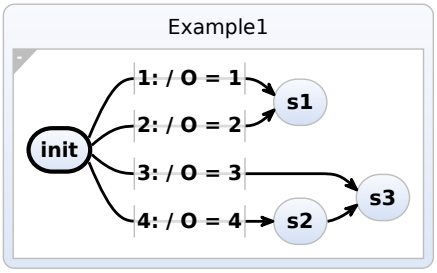
- we define the concept of vertex and edge graph order (Sec. 3.1);
- we adapt crossing minimization for proper layered *ordered* graphs (Sec. 3.2);
- we extend the proposed solution to include *partially* ordered graphs (Sec. 3.3);
- we extend the solution to include backward edges (Sec. 3.4);
- we extend the solution to dangling source vertex ordering in (Sec. 3.5);
- we propose an order metric that can be used to further improve crossing minimization (Sec. 3.6).

```
1  scchart Example1 {
2    output int O
3    initial state init // v1
4    do O = 1 go to s1 // e11
5    do O = 2 go to s1 // e12
6    do O = 3 go to s3 // e13
7    do O = 4 go to s2 // e14
8
9    state s1 // v2
10
11   state s2 // v3
12   go to s3 // e31
13
14   state s3 // v4
15 }
```

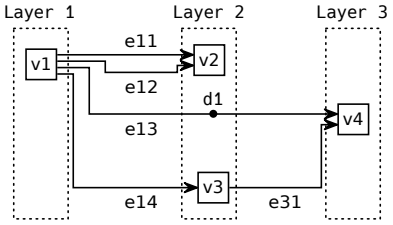
(a) Textual SCChart input file



(b) SCChart synthesized from textual input with Sugiyama layout not considering order



(c) SCChart considering order (this work)



(d) The underlying graph

Figure 1.1: An SCChart with an underlying ordered graph $G = (V, E)$ with $V = \langle v_1, v_2, v_3, v_4 \rangle$ and $E = \langle \langle e_{11}, e_{12}, e_{13}, e_{14} \rangle, \langle \rangle, \langle e_{31} \rangle, \langle \rangle \rangle$

The resulting algorithm configurations are discussed and evaluated in Chap. 4. Chap. 5 presents related work and Chap. 6 concludes this paper and presents future.

2 Layered Algorithm for Unordered Graphs

We define a graph $G = (V, E)$, *vertices* $V = \{v_1, \ldots, v_n\}$, *edges* $E \subseteq P \times P$, and *ports* P that the edges are anchored at, where $\mathsf{P}(v)$ is the subset of ports that belong to a vertex v. Conversely, $\mathsf{v}(p) \in V$ denotes the vertex that port p is anchored at. For an edge $e = (p, q)$, $\mathsf{p_s}(e) = p \in \mathsf{P}(v)$ describes the *source port* and $\mathsf{p_t}(e) = q \in \mathsf{P}(w)$ describes the *target port*. For simplicity, we also write $e = (v, w)$ for source vertex v and target vertex w as short form of $e = (p, q)$, $\mathsf{v}(p) = v$, and $\mathsf{v}(q) = w$ if we do not care about the ports. For port p, $\mathsf{type}(p) \in \{\mathsf{src}, \mathsf{tgt}\}$ indicates whether p is a source or target port and $\mathsf{e}(p) \in E$ returns the edge of the port p.

The algorithm places vertices in vertical layers, as seen in Fig. 1.1d, and only routes edges between two layers, i. e. *in-layer edges* are forbidden. As shown in Alg. 1, the algorithm is divided into five phases: cycle breaking, layer assignment, crossing minimization, vertex placement, and edge routing (STT81). The first two phases transform the digraph into a proper layered digraph. In a *layered* graph $G = (V, E, L)$ the set of vertices V is partitioned into m mutually exclusive ordered subsets that represent their layering $L = (L_1, \ldots, L_m)$, with $L_i = \langle v_{L_{i1}}, \ldots, v_{L_{ir}} \rangle$ for a layer of size r. $\mathsf{L}(v) = i$ denotes the layer i of a vertex $v \in V$. A graph is *proper layered* iff for all edges $e = (v, w) \in E$, $\mathsf{L}(w) = \mathsf{L}(v) + 1$ holds. Since this is generally not possible for digraphs, *dummy vertices* and *dummy edges* are added to replace *long edges* that span multiple layers. In Fig. 1.1d, the edge from v_1 to v_4 is a long edge with one dummy vertex d_1 in layer 2. For simplicity reasons we will ignore *label dummy vertices*, which are generated to place labels on edges, since they are handled just as normal dummy vertices. We distinguish between *real vertices* and dummy vertices.

We call vertices with no incoming edges *sources* and vertices with no outgoing edges *sinks*, and define the functions $\mathsf{indegree} : V \to \mathbb{N}$ and $\mathsf{outdegree} : V \to \mathbb{N}$ that return the number of incoming and outgoing edges of a vertex.

Algorithm 1: layered

Input: A digraph $G = (V, E)$
Output: A digraph with edge routes and vertex coordinates

```
1 G = cycleBreaking(G)
2 G = layerAssignment(G)
3 G = crossingMinimization(G)
4 G = nodePlacement(G)
5 G = edgeRouting(G)
6 return G
```

Cycle breaking transforms a given graph into an acyclic one. This problem is commonly

known as the minimum feedback arc set problem and is NP-hard (Kar72). As an example of a cycle breaking strategy the greedy cycle breaking heuristic (ELS93) is shown in Alg. 2.

Algorithm 2: cycleBreaking

Input: A digraph $G = (V, E)$
Output: An acyclic digraph

```
1 G' = G
2 while G' is not empty do
3     // Repeatedly remove all sources and sinks from G' until only connected components c_1, ..., c_n
        remain
4     // For each c_i: find vertex v with maximum outdegree(v) − indegree(v)
5     // Reverse all incoming edges of v
6 return G
```

We call the edges that are reversed in this process *backward edges*. In the following algorithm, they are handled as normal edges. During the edge routing phase, they are reversed to their original direction.

The layer assignment phase creates a proper layered graph by introducing dummy vertices and dummy edges, as seen in Alg. 3.

Algorithm 3: layerAssignment

Input: An acyclic digraph $G = (V, E)$
Output: A proper layered digraph

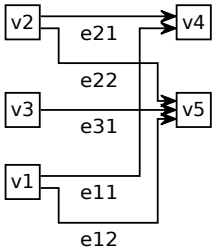
```
1  // Prune graph until only connected components remain
2  G' = G
3  l = longestPath(G)
4  L = {L_1, ..., L_l}
5  while l >= 0 do
6      foreach v in G' do
7          // Assign all sinks of G' to the current layer L_l
8          // Remove all sinks of G'
9      l = l − 1
10 while G contains long edges do
11     foreach long edge e = (v, w) do
12         // Remove long edge from E
13         // Add dummy vertex d to V in layer L(v) + 1
14         // Add edges (v, d) and (d, w) to E
15 return G
```

Crossing minimization uses the proper layered digraph and orders all vertices in their layers and ports on their vertices such that minimal edge crossings are created, as seen simplified in Alg. 4.

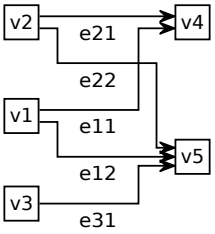
Algorithm 4: crossingMinimization (original)

Input: A proper layered graph $G = (V, E, L)$
Output: A proper layered ordered graph

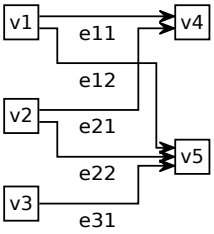
```
1  r = randomSeed // A fixed random seed
2  t = 7 // Thoroughness
3  sweepForward = sweepDirection(r)
4  bestOrder = null
5  for i = 0; i < t; i = i + 1 do
6      G = randomizeLayers(G, r, sweepForward)
7      do
8          foreach L_i ∈ L do
9              minimizeCrossings(L_i)
10     while improved(G);
11     if crossings(G) < crossings(bestOrder) then
12         bestOrder = G
13     sweepForward = ¬sweepForward
14 return bestOrder
```



(a) Local minimum occurs with thoroughness of 1



(b) One optimal solution, achieved by a thoroughness greater than 1



(c) Another optimal solution that preserves the vertex and edge order

Figure 2.1: A graph with a local crossing minimum

Since the crossing minimization problem is NP-hard and remains NP-hard on bipartite graphs (GJ83), a heuristic that includes ports (SFvHM10) is used. Crossing minimization consists of several runs to prevent local minima bounded by the thoroughness value t. Seeded random values are used to guarantee the same diagram for the same graph. For the first run it is randomly decided whether the layers are traversed beginning with the first or the last layer. We call this the *sweep direction* and distinguish between a forward sweep and a backward sweep. Moreover, the random seed is used to reorder all independent vertices and ports, i. e. all sources or sinks and their ports, via randomizeLayers. In this algorithm, edges are ordered via the ports they are anchored at. Since edges only connect ports in neighboring layers, ordering the ports is enough and edge order is defined by them.

Random permutation of the first or last layer is applied to prevent local minima. Fig. 2.1a shows a graph layouted with a thoroughness value of 1. The random initial order yields a local minimum that can be resolved by using a higher thoroughness value or by permuting

the first layer, as seen in Fig. 2.1b and Fig. 2.1c. The thoroughness value of 7 proved to be sufficient to prevent local minima even for large graphs in its implementation in the Eclipse Layout Kernel (ELK)[1].

We call the current layer the *fixed layer* and the next layer (in case of a forward sweep L_{i+1}, else L_{i-1}) the *free layer*. At this point we consider the layers ordered and use a crossing minimization strategy, such as the barycenter heuristic (SFvHM10), to order the vertices and ports in the free layer and continue to do so with the next layer while sweeping forward and backward until no improvement can be found. For each run the resulting edge crossings are counted efficiently by using the order of their ports, as described by (BMJ04). The run that yields the smallest number of crossings defines the order of the vertices in each layer and the order of the ports on each vertex.

To evaluate our approach we use the Barycenter method proposed by Sugiyama et al. to minimize the crossings. However, any approach that does not change the vertex order if it is crossing minimal would work here, such as the median heuristic (EW86) or any approach that sweeps through the layers and counts crossings to compare the result.

[1] https://www.eclipse.org/elk/

3 GRAPH ORDER CROSSING MINIMIZATION

As explained earlier, a key difference between the standard layered approach and our proposal is that we consider vertices and edges to be ordered. The next section will formalize this order. This serves as grounding for the subsequent sections, which explain how to produce drawings that aim to reflect that order whenever this is possible without compromising other aesthetic criteria, specifically the number of edge crossings.

3.1 Graph Order

Def. 1 (Ordered Graph)**.** *We define an ordered graph as* $G = (V,E)$ *were* $V = \langle v_1, \ldots, v_n \rangle$ *is the ordered set of vertices and* $E = \langle \langle e_{11}, \ldots, e_{1k_1} \rangle, \ldots, \langle e_{n1}, \ldots, e_{nk_n} \rangle \rangle$ *is the ordered set of ordered sets of* k_i *outgoing edges for each vertex* v_i*. E implicitly defines an ordered set of outgoing and incoming ports P at which each edge is anchored.*

An example of an ordered graph that follows Def. 1 can be seen in Fig. 1.1. A proper layered ordered graph $G = (V,E,L)$ is defined analogously. We define $\mathsf{o} : V \cup E \cup P \to \mathbb{Z}$ (see Def. 2) as the function that assigns a *graph order value* to vertices, edges, and ports.

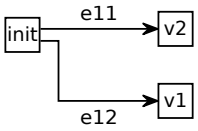
Def. 2 (Graph Order o)**.** $\mathsf{o}(v) = n$ *if* $v \in V$ *is the nth vertex in the graph. Analogously,* $\mathsf{o}(e) = n$ *if* $e \in E$ *is the nth edge in the graph and* $\mathsf{o}(p) = \mathsf{o}(\mathsf{e}(p))$ *for port* $p \in P$*.*

Note that we do not make any further assumptions on where the graph order is coming from. In our motivating example of textually specified SCCharts (see Fig. 1.1a), the graph order is induced by the textual order of states (vertices) and transitions (edges) in the input file. However, the graph order does not have to come from a textual input. For example, the graph order may also be derived somehow from a drawing (sketch) of the graph. In that case, the method presented here can be used to produce a layered drawing of a graph that tries to preserve the "gestalt" of the sketch.

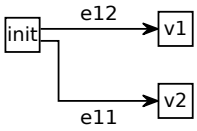
The graph order specifies orders for vertices and edges, as expressed by o. Ideally, this graph order is also reflected in the drawing of the graph, as is the aim of this work. However, this is not always possible, at least not simultaneously for both vertices and edges, as they may sometimes induce conflicting orderings, as illustrated in the example in Fig. 3.1. Actually one might argue that such cases could and should be avoided, e. g. when writing a textual SCCharts specification, but we still want to be able to handle such cases. We, therefore, distinguish the graph order on vertices and edges from the *drawing order* defined by a vertex order $\prec_v : V \times V$ and a port order $\prec_p : P \times P$. We also introduce a flag $\mathsf{prioEdgeOrder} = \neg\mathsf{prioVertexOrder}$ to express whether vertex or edge order is prioritized.

```
digraph {
  rankdir=LR;
  init;
  v1;
  v2;
  init → v2;
  init → v1;
}
```

(a) Textual dot file



(b) Prioritize edge order over vertex order (prioEdgeOrder)



(c) Prioritize vertex order over edge order (prioVertexOrder)

Figure 3.1: We can either prioritize edge or vertex order. This may yield different drawings depending on the graph order.

Def. 3 (Vertex Order $\prec_v$ for ordered graphs)**.** *For $v, w \in L_i$ for some layer $L_i \in L$, we define $\prec_v$ such that $v \prec_v w$ holds iff one of the following cases applies:*

1. $\mathsf{prioVertexOrder} \wedge \mathsf{o}(v) < \mathsf{o}(w)$.

   I. e. vertices are ordered by their graph order.

   For example in Fig. 1.1, we have $v_2 \prec_v v_3$

2. $\mathsf{prioEdgeOrder} \wedge \mathsf{p_s}(\mathsf{getFirstEdge}(\mathsf{v})) \prec_p \mathsf{p_s}(\mathsf{getFirstEdge}(w))$ *where* getFirstEdge *returns the edge on the first port of the vertex. The first edge is on the incoming port that was deemed smallest by $\prec_p$.*

   I. e. vertices are ordered by their incoming edges and not by the graph order.

Intuitively the port order is defined such that $a \prec_p b$ holds iff $\mathsf{o}(a) < \mathsf{o}(b)$. However, this does not capture that edges with the same target should be bundled together, which motivates the following definition.

Def. 4 (Port Order $\prec_p$)**.** *For ports a and b attached to the same vertex v, $\mathsf{v}(a) = v = \mathsf{v}(b)$, we define $\prec_p$ such that $a \prec_p b$ holds iff one the following cases applies:*

1. $\mathsf{type}(a) = \mathsf{type}(b) = \mathsf{src} \wedge \mathsf{v}(\mathsf{p_t}(\mathsf{e}(a))) = \mathsf{v}(\mathsf{p_t}(\mathsf{e}(b))) \wedge \mathsf{o}(a) < \mathsf{o}(b)$.

   I. e. means outgoing ports that connect to the same target vertex are ordered by the graph order of their edges.

2. $\mathsf{type}(a) = \mathsf{type}(b) = \mathsf{src} \wedge \mathsf{v}(\mathsf{p_t}(\mathsf{e}(a))) = w \neq u = \mathsf{v}(\mathsf{p_t}(\mathsf{e}(b))) \wedge \mathsf{o}(\mathsf{getMinEdge}(v,w)) < \mathsf{o}(\mathsf{getMinEdge}(v,u))$ *where* $\mathsf{getMinEdge} : V \times V \to E$ *returns the edge with the minimum graph order* o *between two vertices.*

   I. e. outgoing edges that do not connect to the same target vertex are ordered by the minimal edge order of their target.

   E. g. this reduces unnecessary edge crossings in Fig. 3.2 since placing e_{43} below e_{42} would always produce a crossing and bundles edges with the same target.

3. $\mathsf{type}(a) = \mathsf{type}(b) = \mathsf{tgt} \wedge \mathsf{p_s}(\mathsf{e}(a)) \prec_p \mathsf{p_s}(\mathsf{e}(b))$.

   I. e. incoming ports are sorted as the corresponding source port of their edge. This is needed to prevent unnecessary crossings since the source ports are already correctly ordered by $\prec_p$.

We note that for edge e and f, $\mathsf{o}(e) < \mathsf{o}(f)$ does not imply that $\mathsf{p_s}(e) \prec_p \mathsf{p_s}(f)$ or $\mathsf{p_t}(e) \prec_p \mathsf{p_t}(f)$.

3.2 Crossing Minimization for Proper Layered, Ordered Graphs

Our goal is to order all ports and vertices before crossing minimization. We change crossingMinimization in Alg. 4 such that the *bestOrder* is initialized with the input graph G, the first run starts with a forward sweep, and the second run with a backward sweep, as seen in Alg. 5. For these first two runs randomizeLayers is not executed since the graph is already ordered.

Algorithm 5: crossingMinimization (new)

Input: A proper layered graph $G = (V, E, L)$
Output: A proper layered ordered graph

```
1  r = randomSeed // A fixed random seed
2  t = 7 // Thoroughness
3  sweepForward = true
4  foreach v ∈ V do
5      sort(P(v), ≺_p)
6  foreach L_i ∈ L do
7      sort(L_i, ≺_v)
8  bestOrder = G
9  for i = 0; i < t; i = i + 1 do
10     if i > 1 then
11         G = randomizeLayers(G, r, sweepForward)
12     do
13         foreach L_i ∈ L do
14             minimizeCrossings(L_i)
15     while improved(G);
16     if crossings(G) < crossings(bestOrder) then
17         bestOrder = G
18     sweepForward = ¬sweepForward
19 return bestOrder
```

To order all vertices and ports as the graph model dictates it, one has to first order all ports on each vertex, as seen in Alg. 5, lines 4/5, and then order all vertices (see lines 6/7).

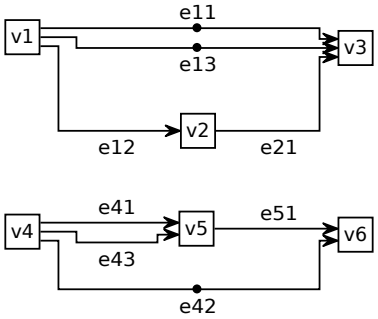


Figure 3.2: Two graphs with long edges and multiple edges to the same target. Edges with the same target are grouped together to reduce potential edge crossings, (dummy vertices marked as black circles).

3.3 Partially Ordered Graphs

When long edges are introduced, the graph definition changes and dummy vertices and edges are added that have no graph order. A properly layered, partially ordered graph G is defined as the tuple (V', E', L') with $V' = (V, V_D)$ with $V_D = \{d_1, \ldots, d_n\}$ as the set of dummy vertices, $E' = (E_L, E_D)$ with E_L the ordered set of edges in which long edges (v, w) are replaced by shortened long edges (v, d_i) and $E_D = \{e_{d_1}, \ldots, e_{d_n}\}$ the set of dummy edges. As before, P' contains the ports of the corresponding edges of E'. $L' = (L'_1, \ldots, L'_n)$ with $L'_i = (L_i, L_{D_i})$ consists of the ordered part L_i and the set of dummy vertices L_{D_i} in that layer. For the example in Fig. 1.1, we add dummy vertex d_1 and replace long edge $e_{13} = (v_1, v_4)$ by edges (v_1, d_1) and (d_1, v_4). This results in the proper layered, partially ordered graph $G = (V', E', L')$ with

- $V' = (\langle v_1, v_2, v_3, v_4 \rangle, \{d_1\})$
- $E' = (\langle\langle e_{11}, e_{12}, (v_1, d_1), e_{14} \rangle, \langle\rangle, \langle e_{31} \rangle, \langle\rangle\rangle, \{(d_1, v_4)\})$
- $L' = ((\langle v_1 \rangle, \emptyset), (\langle v_2, v_3 \rangle, \{d_1\}), (\langle v_4 \rangle, \emptyset)$

Dummy vertices and edges are originally not part of the graph and have, therefore, no derived graph order. We extend o such that $\mathsf{o}(e_{d_i}) = \mathsf{o}(e_{kj})$ for a dummy edge e_{d_i} if e_{kj} is the original long edge the dummy edge was created for. Note that a dummy vertex has no defined graph order value.

We have to change cases 1 and 2 of Def. 4 to also handle long edges. Instead of comparing the target vertex, the *long edge target vertex*, which corresponds to the real vertex the edge eventually connects to, of each port is compared. As seen in Fig. 3.2, this orders e_{11} and e_{13} next to each other, the same way as e_{41} and e_{13} are ordered together.

Def. 3 also has to be changed. The condition in case 1 changes to $(\mathsf{prioEdgeOrder} \vee c \in V_D \vee u \in V_D) \wedge \mathsf{v}(\mathsf{p_s}(\mathsf{getFirstEdge}(\mathsf{v}))) \prec_v \mathsf{v}(\mathsf{p_s}(\mathsf{getFirstEdge}(w)))$. Dummy vertices are compared to other vertices using the incoming edges.

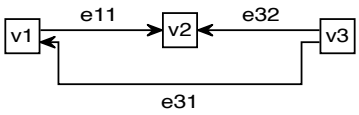


Figure 3.3: Backward edges might produce a not so obvious consistent ordering.

3.4 Backward Edges in Ordered Graphs

Backward edges are in most cases already handled by the algorithm. For a consistent drawing style, we want to place backward edges below normal ones and change Def. 4 case 1 such that this is the case, and change getMinEdge in case 2 such that the graph order of backward edges is not considered here since they originate from a different vertex.

One has to keep in mind that the algorithm orders backward edges as if their source and target are switched, as seen in Fig. 3.3. If one takes a look at the edges e_{31} and e_{32}, these edges seem unordered compared to their source vertex v_3. Since the algorithm sees only edge e_{31} as an outgoing edge of v_1 this is still a consistent ordering.

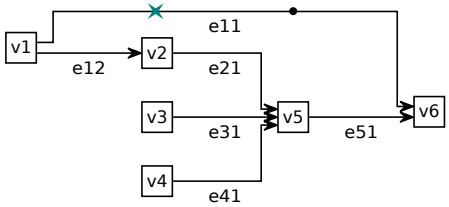
Backward edges are often placed below normal edges, even without the addition proposed above, since they are generally declared after the forward edges and have a higher graph order. If this is not the case, the proposed changes prevent that backward edges cause seemingly non-existent order violations based on Def. 4.

3.5 Dangling Source Vertices in Ordered Graphs

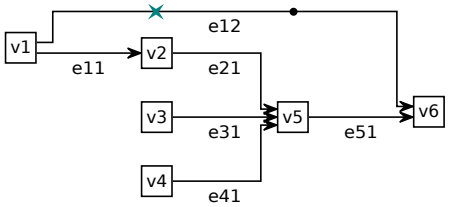
Dangling source vertices are vertices that have no incoming edges but are not in the first layer, e. g. v_3 and v_4 in Fig. 3.4a. Such vertices can be compared to all other real vertices, however, not to dummy vertices. Since dangling source vertices have no incoming edges, this leaves no viable way to compare them. Therefore, the transitively defined ordering has to be respected when sorting layers with such vertices. If this leaves no clear position for a dangling source vertex, one has to decide whether they are above or below dummy edges. As seen in Fig. 3.4, both options might be undesired.

In Fig. 3.4a sorting the dummy vertex above vertex v_3 and v_4 results in a consistent graph. However, for the slightly adjusted graph in Fig. 3.4b this is no longer correct and a conflicting order results in placing edge e_{12} above e_{11}, since the desired order in Fig. 3.4c produces an edge-crossing. The first graph could not achieve a correctly ordered drawing if the dummy vertex would be ordered below v_4 and the second one would also do that (see Fig. 3.4b) or produce a crossing (see Fig. 3.4c). We arbitrarily propose to order dummy vertices above normal vertices (dummyVerticesAbove). However, Fig. 3.4e requires a dummy vertex to be sorted between dangling source vertices, therefore, any fixed ordering relation may be faulty. We extend Def. 3 to the following:

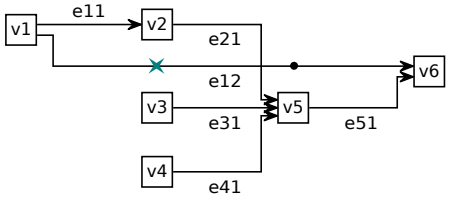
Def. 5 (Vertex Order $\prec_v$ for partially ordered graphs)**.** *For $v, w \in L_i$ for some $L_i \in L'$, we define $v \prec_v w$ iff one of the following cases applies:*



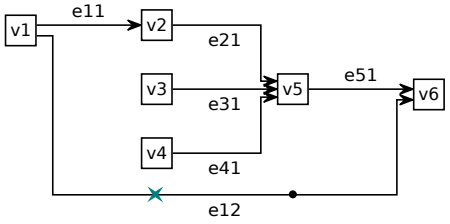
(a) Placing the marked dummy vertex above v_3 and v_4 yields a consistent order



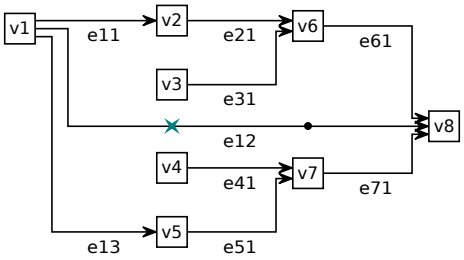
(b) Placing the marked dummy vertex above v_3 and v_4 yields a conflicting order, since edge e_{12} is above e_{11}



(c) Placing the marked dummy vertex above v_3 and v_4 yields an edge-crossing



(d) Placing the marked dummy vertex under v_3 and v_4 yields a consistent order



(e) In this case the marked dummy vertex has to be placed between v_3 and v_4 to yield a consistent order

Figure 3.4: Dummy vertex placement in relation to dangling source vertices.

1. *An existing transitive ordering has to be respected.*

   E. g. if $v \prec_v u$ and $u \prec_v w$, then $v \prec_v w$ holds.

2. $(\mathsf{prioVertexOrder} \vee \mathsf{incoming}(v) = \mathsf{incoming}(w) = 0) \wedge \mathsf{o}(v) < \mathsf{o}(w)$.

   Two dangling source vertices have to also be compared by their graph order.

3. $(\mathsf{prioEdgeOrder} \vee v \in V_D \vee w \in V_D) \wedge \mathsf{v}(\mathsf{p_s}(\mathsf{getFirstEdge}(\mathsf{v}))) \prec_p \mathsf{v}(\mathsf{p_s}(\mathsf{getFirstEdge}(w)))$.

   Since dummy vertices have no graph order, we always have to use the incoming edges to compare them to other vertices.

4. $(\mathsf{prioEdgeOrder} \vee v \in V_D \vee w \in V_D) \wedge$
   $\mathsf{dummyVerticesAbove}\,?\,\mathsf{indegree}(w) = 0 : \mathsf{indegree}(v) = 0$.

   Dummy vertices are sorted above or below dangling source vertices as one alternative since there is no way to compare these two kinds of vertices.

3.6 A Graph Order Metric

The defined relations $\prec_v$ and $\prec_p$ serve as a metric to decide how good a graph is ordered. This metric can be used as a secondary criterion during crossing minimization. To do this, line 16 in Alg. 5 is changed by adding the port or vertex order violations multiplied by a weight w_v and w_p, so that the condition becomes:

$$\begin{aligned} & w_p \cdot \mathsf{portOrderViolations}(G) \\ & + w_v \cdot \mathsf{vertexOrderViolations}(G) + \mathsf{crossings}(G) \\ < \; & w_p \cdot \mathsf{portOrderViolations}(bestOrder) \\ & + w_v \cdot \mathsf{vertexOrderViolations}(bestOrder) \\ & + \mathsf{crossings}(bestOrder) \end{aligned}$$

where portOrderViolations and vertexOrderViolations count the number of port and vertex order violations for a partially ordered graph. These weights express how many order violations are as important as an edge crossing. E. g. $w_v = w_p = 0.1$ means that 10 vertex or port order violations are as important as an edge crossing. If ordering the graph before crossing minimization is not enough since the graph is conflicting, this metric can be used to get drawings that comply more often with the $\prec_v$ or $\prec_p$ metric. Using this, Fig. 3.4b could be drawn as desired (see Fig. 3.4d).

Note that this approach only reduces the probability of NONO cases in conflicting models but does not fully eliminate them.

4 EVALUATION

We compare nine different algorithm configurations, as seen in Tab. 4.1. Note that N_V and N_E produce the same graph but are differently evaluated regarding their violations of the graph order, as described in Sec. 4.2.

We consider 54 SCCharts that were developed by humans. SCCharts models may consist of concurrent more than one concurrent region with states and edges between them. Each region has its own graph. For each model we, therefore, might solve several graph drawing problems. The chosen models have two to 72 vertices per region and up to 310 vertices per model with an average of 44 vertices per model (including dummy vertices). There are from one to 16 vertices per layer. The edge density to adjacent layers is two to 73 with an average of 9. The average vertex degree is between zero and seven.

In general, we limit our approach to graphs with a human readable size. SCCharts with more than 100 states in the same hierarchy level in a region and, therefore, 100 vertices and several edges are not what the proposed approach was designed for since the whole diagram might not be readable regardless whether vertices are ordered or not.

For all graphs the ordering step is done in a fraction of a millisecond and is significantly quicker and less complex than crossing minimization in general. The layout direction is set the RIGHT and the dummy vertices are sorted above normal vertices (dummyVerticesAbove). How a value for w_v and w_p is chosen is described in the following.

4.1 Weighted Ordering with w_v and w_p

Tab. 4.2 illustrates the effects of varying w_v and w_p on edge crossings and on the number of fully ordered drawings of the 54 graphs. Increasing the weight of $\prec_v$ and $\prec_p$ during crossing minimization tends to increase the number of correctly ordered graphs at the cost of edge crossings, but it cannot always find an ordering with minimal order violations. The reasons

w_p	w_v	N	V	E
0	0	N_V, N_E	V	E
>0	0		$V_{w_p,0}$	$E_{w_p,0}$
0	>0		V_{0,w_v}	V_{0,w_v}
>0	>0		V_{w_p,w_v}	E_{w_p,w_v}

Table 4.1: N (unordered), V (prioVertexOrder), E (prioEdgeOrder). Overview and encoding of the evaluated algorithmic alternatives. Columns differ in whether vertices or edges are prioritized, rows differ in the weights assigned to vertex/port order violations relative to edge crossings, which carry a weight of 1.

for this is that the barycenter heuristic (or any other commonly used approach) used during crossing minimization does not focus on the order but on crossing minimization. If no run yields the ordered solution, it cannot be chosen, even though it would be chosen if it occurred, based on the weights w_v and w_p.

If one wanted to maintain the order and disregard the crossings, it would be necessary to develop a different crossing minimization strategy or to use existing strategies to enforce vertex order (see Chap. 5).

For differently weighted V_{0,w_v} approaches with $w_v > 0$, randomization seems to have the most visible effect. The number of order violations increases instead or decreases with higher weights for w_v and w_p. The reason for this is randomization (see Sec. 4.4). When comparing $\mathrm{V}_{0,0.001}$ to $\mathrm{V}_{0,0.5}$, the number of violations decreases, no additional edge crossings are created, and we have more fully ordered drawings. Again, this happens because of randomization and because most of the order violations are produced by only four conflicting models. In this case, this produced randomly worse results. As mentioned in Sec. 3.6 and further explained in Sec. 4.4, we cannot make any ordering guarantees for conflicting models.

4.2 Quantitative Evaluation

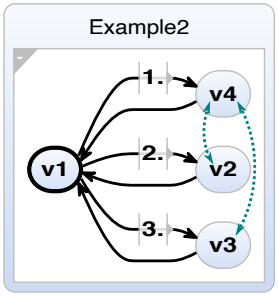
The number of drawing order violations for the different approaches can be seen in Tab. 4.2. N_V serves as a baseline for all approaches that have prioVertexOrder set (i. e. V and V_{w_p,w_v}), N_E serves as a baseline for approaches with prioEdgeOrder (i. e. E, E_{w_p,w_v}). The resulting drawing of N_V and N_E is the same, we only count the $\prec_v$ violations by comparing the graph order of real vertices in the N_V case and use the edge graph order for N_E, as introduced in Sec. 3.5. All approaches that prioritize edge order have, therefore, fewer vertex order violations.

Fig. 4.1 visualizes how $\prec_v$ and $\prec_p$ order violations are counted for the different approaches. In Fig. 4.1a, the shown vertex order violations are only counted for all prioVertexOrder approaches since prioEdgeOrder approaches order vertices by their incoming edges. In Fig. 4.1b, we see two edge order violations. Furthermore, note that the V approach would not produce this drawing but the $\mathrm{V}_{0,0.001}$ approach would. The V would initially order the ports and vertices without creating violations (e. g. edge 1 above edge 2 and 3). If crossing minimization starts, the vertices v_2 to v_4 are in a free layer and their order is changed to comply with the port order to not produce additional crossings. For the $\mathrm{V}_{0,0.001}$ approach this creates two violations. The second run would then yield the drawing in Fig. 4.1b since v_2 to v_4 are in the fixed layer for a backward sweep. The resulting drawing has no vertex order violations and no crossings and is, therefore, better than Fig. 4.1a under the $\mathrm{V}_{0,0.001}$ approach.

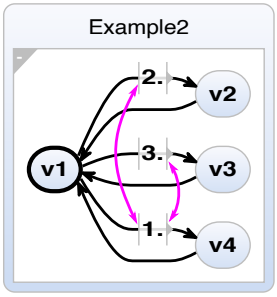
The two approaches V_{w_p,w_v} and E_{w_p,w_v} that use the $\prec_v$ and $\prec_p$ metrics as a secondary criterion to edge crossings have the fewest total number of order violations for their respective approaches, as seen in Tab. 4.2. Note that all approaches significantly decrease the number of order violations with respect to their baseline. Many models have no violations at all. For two models the total order violations sometimes increased compared to the N_V or N_E approach.

	$\prec_v$	$\prec_p$	Fully ordered drawings	Crossings
N_V	250	695	1	26
V	143	91	23	32
$V_{0.001,0}$	143	91	23	32
...	...	...	...	...
$V_{100,0}$	143	91	23	32
$V_{0,0.001}$	91	173	24	28
$V_{0,0.01}$	91	173	24	28
$V_{0,0.1}$	91	173	24	28
$V_{0,0.5}$	44	172	24	28
$V_{0,1}$	81	184	25	39
$V_{0,10}$	64	159	26	82
$V_{0,100}$	72	169	26	112
$V_{0.001,0.001}$	122	96	24	28
$V_{0.01,0.01}$	122	96	24	28
$V_{0.1,0.1}$	122	96	24	28
$V_{0.5,0.5}$	95	78	26	36
$V_{1,1}$	102	48	29	93
$V_{10,10}$	129	12	29	149
$V_{100,100}$	129	12	29	149
N_E	45	695	1	26
E	27	91	31	32
$E_{0.001,0}$	27	91	31	32
...	...	...	...	...
$E_{100,0}$	27	91	31	32
$E_{0,0.001}$	14	93	32	28
$E_{0,0.01}$	14	93	32	28
$E_{0,0.1}$	14	93	32	28
$E_{0,0.5}$	10	88	32	28
$E_{0,1}$	10	101	33	32
$E_{0,10}$	12	101	33	36
$E_{0,100}$	10	95	33	34
$E_{0.001,0.001}$	14	92	32	28
$E_{0.01,0.01}$	14	92	32	28
$E_{0.1,0.1}$	14	92	32	28
$E_{0.5,0.5}$	12	78	33	34
$E_{1,1}$	13	52	39	46
$E_{10,10}$	32	20	39	92
$E_{100,100}$	32	20	39	92

Table 4.2: Graph order violations for the metrics $\prec_v$ and $\prec_p$ for w_v and w_p set to 0.001, 0.01, 0.1, 0.5, 1, 10, and 100 for their respective approaches. Lines for $V_{w_p,0}$ and $E_{w_p,0}$ are omitted since they did not change with varying w_p. *Fully ordered drawings* describes the number of models that have no order violations in any part of their model. *Crossings* describes the total number of edge crossings in all regions of all 54 models with the corresponding algorithm.



(a) A drawing with two $\prec_v$ violations for a prioVertexOrder approach and none with prioEdgeOrder.



(b) A drawing with two $\prec_p$ violations.

Figure 4.1: Vertex order violations are shown in teal and edge order violations in magenta.

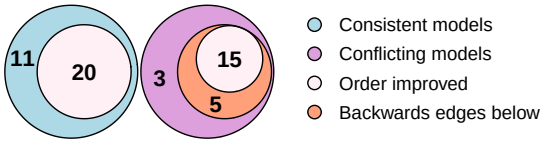


Figure 4.2: Changes of the 54 evaluated models

This happens since these models are conflicting, the initial drawing order is not crossing-minimal, and randomizeLayers (see Alg. 5) can in some cases be randomly better or worse (see Sec. 4.4).

4.3 Qualitative Evaluation

The V approach is evaluated compared to the normal approach. The results can be seen in Fig. 4.2. Of these 54 models, 31 were consistent and 23 were conflicting. 20 of the consistent models improved the order and not just placed backward edges below normal ones. This can also be an improvement if it introduces a consistent drawing style but we chose to distinguish it from a real drawing order improvement. Three of the models that had conflicts did not change in the conflicting regions. 20 of the models that had conflicts nonetheless improved the overall layout. Of these 20 models, 15 had order improvements, the rest just placed backward edges below normal edges. All three conflicting models that did not improve, were, by coincidence, ordered as good as possible.

For all weighted order approaches we set $w_v = w_p = 0.001$ for this comparison. When comparing the 54 models $V_{0,0.001}$ did not change the layout and E and $E_{0,0.001}$ did change one graph to favor edge order when compared to the V layout presented above. Six layouts changed for all other approaches. These changes were vertex movements that created fewer vertex order violations but did not improve the overall layout significantly or a mirroring of part of the graph to reduce vertex order violations. However, with respect to the desired metric, they were better. The $E_{0.001,0.001}$ approach would rather violate the vertex order than

the edge order and did, therefore, more edges in these six models when compared to the V approach. When comparing $V_{0,0.001}$ to V it becomes imminent that the vertex order is here more important than the edge order. Of the 54 models 16 changed their vertex order and eleven of these models broke the correct edge order to order the vertices correctly, as seen for example in Fig. 3.1c. If this is desired, this configuration should be chosen. The $V_{0.001,0.001}$ approach also favored the vertex order over the edge order but only in three of the ten models that improved regarding vertex order. Additionally to the six models, $E_{0,0.001}$ and $E_{0.001,0.001}$ also improved the order in the same model E and $E_{0.001,0}$ improved regarding edge order violations.

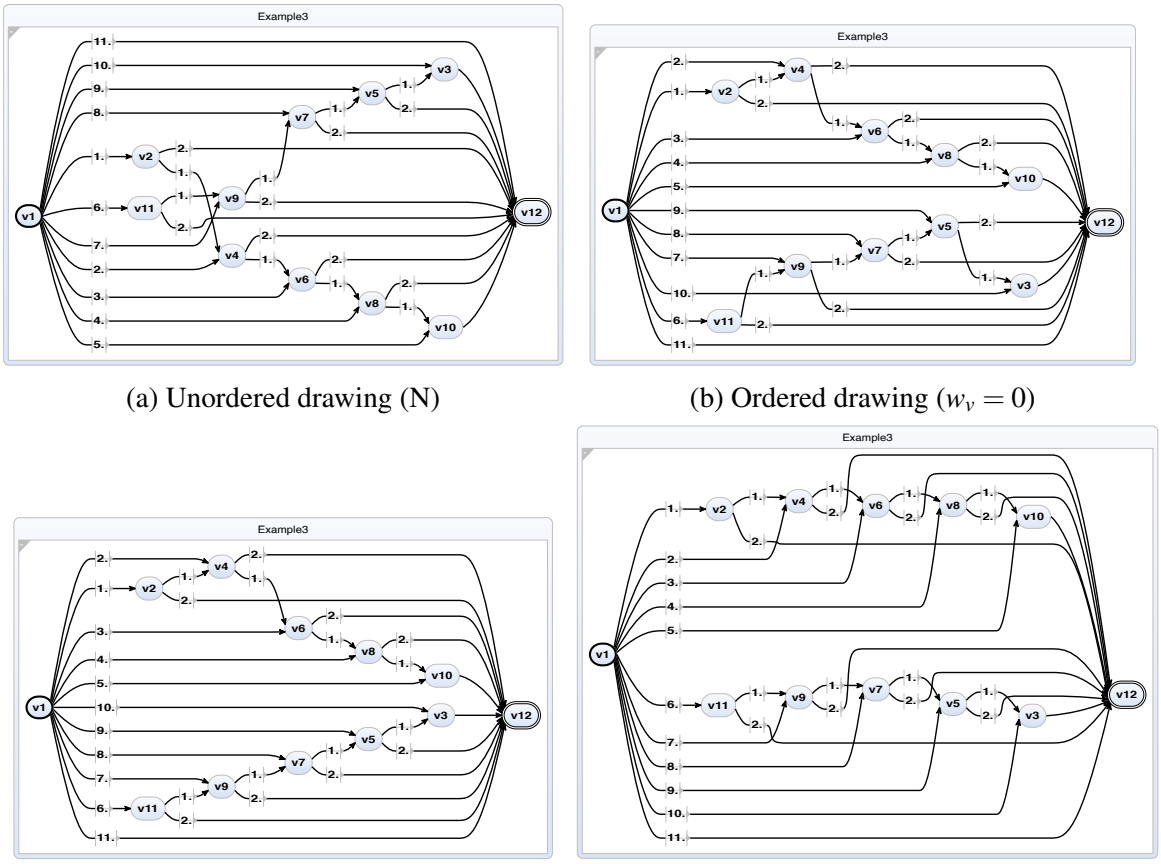
4.4 Influence of Randomization

Tab. 4.2 shows that the N (unordered) approach has fewer total crossings (26) than all other (ordered) approaches, even though we argue that ordering vertices and ports before crossing minimization should not necessarily increase the number of crossings relative to the traditional approach. As it turns out, four of the 54 models have a different number of edge crossings than the N approach. One has one crossing less (from one to zero crossings), two have one crossing more (from zero to one crossing), and another one has either one or three additional crossings depending on the approach (from 15 to 16 or 18 crossings) in some regions of some models. The reason for this is randomization.

Randomization still has an influence on the quality of the solution for some kind of graphs. If a graph is conflicting or the drawing order produces additional crossings, randomizeLayers is still used to create the drawing, as seen in Alg. 5. Therefore, these graphs can by coincidence produce more or fewer edge crossings or order violations.

In the ordered approach the first two runs do not randomize the order and use the graph order for a forward and backward sweep. The order after these two sweeps is not the same order the unordered approach uses for their first run. Therefore, the vertex and port order is already different to the N approach before randomization takes place. Since any kind randomization can lead to different results, some of these runs can randomly be a local minimum or no longer be a local minimum, and result in more or fewer crossings or order violations. Only increasing the thoroughness reduces the probability of additional crossings but does not solve this anomaly.

It would be possible to eliminate this anomaly by first performing the traditional crossing minimization based on the unordered graph, followed by the crossing minimization proposed here, and then comparing the results. However, if one would really want to spend more computation time on crossing minimization we would recommend to increase the thoroughness value instead. I. e. a thoroughness value of 14 and $w_v = w_p = 0.001$ yields just 23 crossings for all ordered approaches.



(a) Unordered drawing (N)

(b) Ordered drawing ($w_v = 0$)

(c) Ordered drawing ($w_v = 0.001$)

(d) Ordered drawing ($w_v = 100$)

Figure 4.3: Drawings via different approaches of part of an evaluated, conflicting model with obfuscated names and omitted edge labels that shows that a drawing might get worse if the underlying graph is carelessly constructed.

4.5 Evaluation

If one favors the edge order, E or E_{w_p,w_v} are the recommended approaches. For SCCharts E_{w_p,w_v} is recommended to still get better results if E falls back to a random starting permutation. If one expects the user to change the vertex order in the model file to create the desired layout, V_{0,w_v} should be used since it can be used to enforce vertex positions. A vertex order influence of 0.001 can be used to get layouts without additional crossings. All prioVertexOrder approaches are controllable by the textual vertex order since the vertex graph order has no semantics. In the SCCharts case, edge order seems to be more important for developers in most cases. In some models the developer did not intend to specify a vertex order or the model is designed carelessly, as seen in Fig. 4.3d. For SCCharts, an edge order is more often intentionally specified since it carries semantic if the conditions are not exclusive. The edge order can be changed but this might need more work than just copying a vertex to a different position in the model.

The proposed approach performs especially good for graphs that a tree-like with a final vertex were everything connects to (which might have a feedback loop) or several feedback loops to the root vertex or another central vertex. Without edge graph order, the different routes trough the tree are not ordered by their priority (e. g. the edge with priority 1 is on top, the edge with priority 2 is below, ...), as the secondary notation suggests, but randomly, which highly irritates the user and impedes understandability since “true” and “false” cases might change sides. As Fig. 4.3 shows, this is not possible for all graphs. Big models benefit from ordered edges and vertices since it is easier to identify the correct vertex or edge one is interested in if one can rely on the ordering to be correct.

One of the main reasons for conflicting graph orders are just too many connections. A highly connected SCChart tends to be conflicting since drawing it crossing-free is often not possible and in most cases the graph order is not crossing-minimal (see Fig. 4.3). However, since the models were not developed with graph order in mind, this is expected to improve in the future. Another option is that the developer did not design the SCChart carefully (see Fig. 3.1 or Fig. 4.3) and did not care about a reasonable order in their SCChart since it did most likely not change the layout, or the developer might not care at all since it is typically not part of a homework assignment to create a well layouted model but rather just a functional one. Very specific SCCharts, especially for papers, did change their graph order to try to influence the randomness of the solution to get a better drawing. This, however, tends to create conflicting and, therefore, often bad layouts if one uses the unnatural deranged graph order to infer a layout. We expect the last three reasons to occur less frequently if modelers already see the influence of the graph order on their drawing.

Note that the different layouts are not evaluated regarding their usability, e. g. how easy one can follow edges, find connected vertices, or similar tasks, since we do not expect this to be worse since nearly no new crossings are created, no new backward edges occur, and the change in edge length and edge straightness is minor for the approaches presented in Tab. 4.1. Therefore, we assume that developers can use a sorted SCChart at least as well as an unsorted one.

5 RELATED WORK

There were already several works that aim to identify or produce a human-like layout and operate at the so called NONO principle (Nothing is obviously non-optimal) (KDMW16). Kieffer et al. use this principle and human participants to identify aesthetic criterions and goals for a human-like orthogonal layout algorithm. Purchase et al. (PAK$^+$20) take this further and try to identify layouts that are obviously machine made. We tackle similar goals. Fig. 1.1b looks obviously machine made and is a drawing a human would not produce. In contrast to Kieffer et al., we choose to still use the layered algorithm and try to conform with the NONO principle by using the graph order as additional layout information to solve obvious problems instead of developing a whole new algorithm.

The are several extensions to the Sugiyama algorithm. We will discuss some of them that aim to influence the order of vertices and edges.

(GKNV93) propose to order the vertices initially by depth-first or breath-first search to reduce initial crossings. This approach is quite similar to ours. We, however, do not order the vertices to reduce potential crossings, but order them to respect the graph order given by the input and set the resulting vertex and port ordering as the currently best vertex and port ordering. Moreover, Gansner et al. do not consider the edge or port order.

(Wad01) uses constraints to order vertices and to force them on specific positions even if this causes additional crossings. This approach is inherently different. We try to preserve the initial order but still try to minimize crossings if possible. We focus on layout *creation*. Waddle, however, focuses on layout *adjustment* and prioritizes order constraints over crossings. Our ordering does not constrain the solution, but rather creates a better crossing-optimal solution. They use layout adjustment to maintain the mental map, we assume that the graph order in the model file is a representation of the modelers mental map.

(MSW19) aim to produce stable drawings by maintaining a global order of vertices. This global order constrains vertex movement in the same layer. Again, this order will not be changed if additional crossings are produced. Moreover, they do not consider the ordering of edges, since there is at most one edge from one vertex to another vertex (not considering backward edges that are reversed).

(BP90) introduce absolute and relative constraints to fix the order of vertices. This is a viable solution to maintain graph order but this, again, does not prevent additional crossings. One can of course manually set all constraints to produce layouts that maintain the order in most parts, but this is a manual effort. We want to automatically produce drawings that maintain the graph order without causing additional crossings. Again, Böringer and Paulisch do not constrain edges but only vertices, which solves only one of our problems since no dummy vertices can be constrained.

Layout Constraints

We argue that ordering all vertices and edges via constraints is no adequate solution. First, all these constraints have to be solved, which can become rather difficult and increases the layout complexity (BP90). Second, it is unclear what to do if there is no solution. One could of course somehow prioritize constraints and decide which to break, but this seems rather involved. Third, the found solutions may produce unnecessary crossings or drawings that otherwise irritate the user. If one introduces constraints there is no middle ground, they are either followed or not. Balancing the importance of edge crossings against constraints seems rather difficult and would again increase computation time. This does not mean that constraints are not helpful to fix positions of certain vertices, but the additional crossings, which they might cause, are not desired in our use case.

Layout Adjustment Methods

Another option to maintain the graph order are layout adjustment methods (MELS95; Nor96). Instead of creating a layout based on all vertices, the layout is created incrementally. Vertices and edges are added incrementally by graph order. Depending on the layout stability constraints, this may introduce unnecessary crossings. It seems possible to replicate the proposed ordering using layout adjustment methods but it seems difficult to achieve a crossing-minimal solution.

6 CONCLUSION

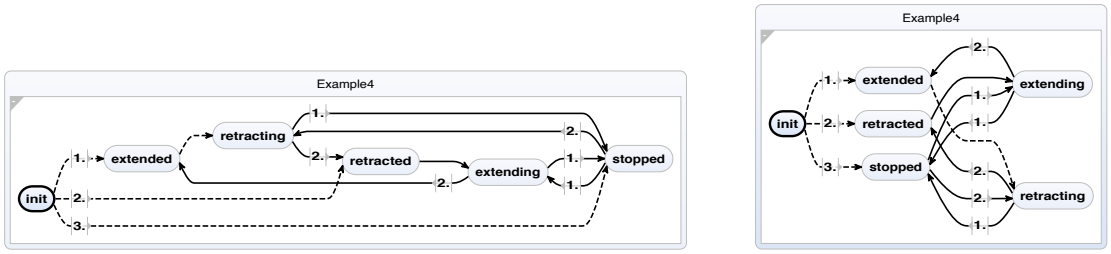
We presented a solution to preserve the graph order by setting an initially best ordering for crossing minimization. This allows us to maintain the graph order without causing additional crossings introduced by local minima other than through coincidence or order constraints for many models.

Including the proposed graph order metric in the crossing minimization step additionally to the crossings as a secondary criterion seems beneficial. Therefore, prioEdgeOrder with weighted vertices and ports (E_{w_p,w_v}) is one potential option for SCCharts. Another one is V since it allows to control the layout without changing the semantic by changing the vertex graph order. Therefore, we make this setting configurable for SCCharts and to evaluate this further.

A graph with different disconnected components can use the graph order metric to order them, as seen in Fig. 3.2. One can either order them by the minimum vertex graph order defined by o or by the mean graph order of the elements of a component. If the components are carefully designed one can assume that if component c_1 should be before component c_2 that for all $v_i \in c_1$ and all $v_j \in c_2$, $\mathsf{o}(v_i) < \mathsf{o}(v_j)$.

Future work on this project should evaluate whether SCCharts that are created in a tool that visualizes the diagram taking the graph order into account results in more consistent models or otherwise changes the way modelers design.

Moreover, a cycle breaking strategy that takes the graph order into account is also worth investigating to have full control over the diagram. Such a cycle breaking together with this work to preserve the graph order would allow to preserve even more of the graph order without additional crossings but by reversing more edges based on the vertex order the developer specified, as seen in Fig. 6.1.



(a) An SCChart with preserved graph order that minimizes backward edges

(b) An SCChart with cycle breaking by vertex graph order

Figure 6.1: Graph order cycle breaking

Bibliography

Wilhelm Barth, Petra Mutzel, and Michael Jünger. Simple and efficient bilayer cross counting. *Journal of Graph Algorithms and Applications*, 8(2):179–194, 2004.

Karl-Friedrich Böhringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 43–51, New York, 1990. ACM.

John Ellson, Emden R. Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. *LNCS*, 2265:594–597, 2002.

Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In *Proceedings of the First International Conference on Computational Graphics and Visualization Techniques*, pages 34–43, 1991.

Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

Peter Eades and Nicholas C. Wormald. The median heuristic for drawing 2-layered networks. Technical Report 69, University of Queensland, Department of Computer Science, 1986.

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

Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. HOLA: human-like orthogonal network layout. *IEEE Trans. Vis. Comput. Graph.*, 22(1):349–358, 2016.

Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995.

Robin JP Mennens, Roeland Scheepens, and Michel A Westenberg. A stable graph layout algorithm for processes. In *Computer Graphics Forum*, volume 38, pages 725–737. Wiley Online Library, 2019.

Stephen C. North. Incremental layout in DynaDAG. In *Proceedings of the Symposium on Graph Drawing*, volume 1027 of *LNCS*, pages 409–418. Springer, 1996.

Helen C Purchase, Daniel Archambault, Stephen Kobourov, Martin Nöllenburg, Sergey Pupyrev, and Hsiang-Yun Wu. The turing test for graph drawing algorithms. In *International Symposium on Graph Drawing and Network Visualization*, pages 466–481. Springer, 2020.

Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD '97)*, volume 1353 of *LNCS*, pages 248–261. Springer, 1997.

Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD '09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010.

Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.

Vance Waddle. Graph layout for displaying data structures. In *Proceedings of the 8th International Symposium on Graph Drawing (GD '00)*, volume 1984 of *LNCS*, pages 98–103. Springer, 2001.