

# Foresighted Graphlayout

*Stephan Diehl*  
*Carsten Görg*  
*Andreas Kerren*

Technischer Bericht Nr. A/02/2000

FR 6.2 - Informatik  
Universität des Saarlandes, Postfach 15 11 50  
D-66041 Saarbrücken, Germany  
Phone: ++49-681-3023915  
{diehl, goerg, kerren}@cs.uni-sb.de

## **Abstract**

In this technical report we introduce the concept of graph animations as a sequence of evolving graphs and a generic algorithm which computes a Foresighted Layout for dynamically drawing these graphs while preserving the mental map. The algorithm is generic in the sense that it takes a static graph drawing algorithm as a parameter. In other words, using an appropriate static graph layouter different kinds of graphs can be animated.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Graph Animations</b>	<b>1</b>
<b>3</b>	<b>Foresighted Layout</b>	<b>2</b>
3.1	Graph Animation Partitionings . . . . .	2
3.2	Strategies for Computing a GAP . . . . .	5
3.3	Reduced Graph Animation Partitionings . . . . .	5
3.4	Algorithm . . . . .	7
<b>4</b>	<b>Implementation</b>	<b>7</b>
<b>5</b>	<b>Examples</b>	<b>8</b>
5.1	Algorithm Animation . . . . .	8
5.2	Resource Allocation . . . . .	10
5.3	Buffered I/O . . . . .	10
<b>6</b>	<b>Interactive Graph Animations</b>	<b>12</b>
<b>7</b>	<b>Conclusion</b>	<b>12</b>

# List of Figures

1	Ad-hoc and Foresighted Layout of the intermediate and final NFA for $(a b)^*$ . . . . .	9
2	Foresighted and Ad-Hoc Layout for deadlock situation. . . . .	10
3	Different graph animations of buffered I/O example. . . . .	11

# 1 Introduction

Most work on graph drawing addresses the problem of laying out a single, static graph. Algorithms have been developed for different classes of graphs (trees, dags, digraphs, ...) and different aesthetic criteria, like minimizing crossings and bends or maximizing symmetries [1, 6]. But the world is full of dynamic graphs, e.g. animations of graph algorithms or algorithms which work on pointered data structures, dynamic visualisations of resource allocation in operating systems and project management, network connectivity and the constantly changing hyperlink structure of the web.

Dynamic graph drawing addresses the problem of laying out graphs which evolve over time by adding and deleting edges and nodes. This results in an additional aesthetic criterium known as “preserving the mental map” [7].

The ad-hoc approach is to compute a new layout for the whole graph after each update using those algorithms developed for static graph layout. In most cases this approach produces layouts which do not preserve the mental map. The common solution is to apply a technique known from key-frame animations called inbetweening to achieve “smooth” transitions between subsequent graphs, i.e. animations show how nodes are moved to their new positions. This approach yields decent results if only a few nodes change their position or whole clusters are moved without substantially changing their inner layout. But in most cases the animations are just nice and do neither convey much information nor help to preserve the mental map. Incremental algorithms try to change the layout just as far as to accommodate the update. Unfortunately, in the worst case they have to compute the layout of the whole graph.

In this technical report we present a totally different approach. Given a sequence of  $n$  graphs we compute a global layout which induces a layout for each of the  $n$  graphs. A unique feature of this approach is that once they are drawn on the screen neither nodes nor the bends of edges change their positions in graphs subsequently drawn. Using static graph layouters, which accept fixed node positions as an additional input, it is also possible that only the bends change their positions. We call the algorithm *Foresighted Layout* as it knows the future of the graph, i.e. the next  $n - 1$  modifications.

## 2 Graph Animations

In the following we consider graphs with multi-edges. For this we add unique identifiers to each edge.

### Definition 1 (Graph)

A graph  $g = (V, E)$  consists of a set of nodes  $V$ , a set of edges  $E \subseteq V \times V \times \text{Id}$  and for all  $(v_1, v_2, n), (v'_1, v'_2, m) \in E : n = m \Rightarrow v_1 = v'_1, v_2 = v'_2$ .

We define a graph animation as a sequence of graphs. A graph results from modifications (adding or deleting nodes and edges) of its preceding graph. Usually subsequent graphs in a graph animation share some nodes and edges. But in the worst case each graph can consist of totally different nodes and edges.

**Definition 2 (Graph Animation)**

A **graph animation**  $G$  is a sequence  $G = [g_1, \dots, g_n]$  of graphs with  $G_i = (V_i, E_i)$  and for all  $(v_1, v_2, n) \in E_p, (v'_1, v'_2, m) \in E_r$  with  $1 \leq p, r \leq n : n = m \Rightarrow v_1 = v'_1, v_2 = v'_2$ .

The restriction in this definition ensures that edge identifiers are used consistently in all graphs, i.e. for edges between the same nodes.

### 3 Foresighted Layout

A first approach to layout a graph animation is to compute its super graph and to reuse its layout information for the layout of the individual graphs in the animation.

**Definition 3 (Super Graph)**

Let  $G$  be a graph animation  $G = [g_1, \dots, g_n]$  with  $g_i = (V_i, E_i)$ , then the **super graph**  $\hat{G}$  of  $G$  is defined as  $\hat{G} = (\hat{V}, \hat{E})$  with  $\hat{V} = \bigcup_{i=1}^n V_i$  and  $\hat{E} = \bigcup_{i=1}^n E_i$ .

In general the super graph will be large and there will be much unused space in the layout of each individual graph. To avoid this *Foresighted Layout* constructs on the basis of the super graph a smaller graph by taking into account the live times of the nodes and edges in the graph animation.

**Definition 4 (Live Time)**

Let  $G = [g_1, \dots, g_n]$  be a graph animation and  $\hat{G} = (\hat{V}, \hat{E})$  its super graph where  $g_i = (V_i, E_i)$ . Then  $T(v) = \{i | v \in V_i\}$  are the **live times** of the node  $v \in \hat{V}$  and  $T(n) = \{i | (v, w, n) \in E_i\}$  are the **live times** of the edge identified by  $n$ .

#### 3.1 Graph Animation Partitionings

**Definition 5 (Graph Partitioning)**

Let  $g = (V, E)$  be a graph and  $\tilde{V} \subseteq \mathcal{P}(V)$  and  $\tilde{E} \subseteq \tilde{V} \times \tilde{V} \times \text{Id}$ . A graph  $\tilde{g} = (\tilde{V}, \tilde{E})$  is a **graph partitioning** of  $g$  iff the nodes in  $\tilde{V}$  are disjoint,  $\bigcup_{v \in \tilde{V}} v = V$  and  $(\tilde{v}_1, \tilde{v}_2, n) \in \tilde{E} \Leftrightarrow \exists v_1 \in \tilde{v}_1$  and  $v_2 \in \tilde{v}_2 : (v_1, v_2, n) \in E$ . We call  $\tilde{E}$  the set of edges induced by  $\tilde{V}$ .

In other words,  $\tilde{V}$  is a partitioning of  $V$ . Each node in  $\tilde{V}$  represents one or more nodes from  $V$  and all edges between two nodes in  $V$  are converted into edges between the representatives of the two nodes.

**Definition 6 (Graph Animation Partitioning GAP)**

Let  $G = [g_1, \dots, g_n]$  with  $g_i = (V_i, E_i)$  be a graph animation and  $\widehat{G} = (\widehat{V}, \widehat{E})$  be the super graph of  $G$ . A graph partitioning  $\tilde{g} = (\tilde{V}, \tilde{E})$  of  $\widehat{G}$  where  $\tilde{V} = \{P_1, \dots, P_k\}$  is a **graph animation partitioning** of  $G$  iff  $v, v' \in P_r \Rightarrow T(v) \cap T(v') = \emptyset$ .

We call  $\tilde{g}$  a *minimal GAP* of  $G$ , if there exists no GAP of  $G$  with less nodes.

In a GAP nodes with disjoint live times are grouped together. Unfortunately, the problem of computing a minimal GAP (hence mGAPP) is  $\mathcal{NP}$ -complete. To prove this we first prove the  $\mathcal{NP}$ -completeness of a sets partitioning problem.

**Definition 7 (Minimal Disjoint Sets Partitioning Problem mDSPP)**

Let  $T = \{T_1, \dots, T_n\}$  be a set of finite sets. Compute a minimal partitioning  $P$  of  $T$  such that in each partition there are only disjoint sets, i.e.  $\forall M \in P : S_1, S_2 \in M \Rightarrow S_1 \cap S_2 = \emptyset$ .

**Lemma 1**

mDSPP is  $\mathcal{NP}$ -complete.

*Proof:* We can now show that minimal graph coloring MGC can be reduced to the above partitioning problem and vice versa. As minimal graph coloring is  $\mathcal{NP}$ -complete [5], it follows that the partitioning problem is  $\mathcal{NP}$ -complete.

- Reduce mDSPP to MGC:  
Given a mDSPP we compute an undirected graph  $G = (\{T_1, \dots, T_n\}, \{\{T_i, T_j\} | T_i \cap T_j \neq \emptyset\})$ . After coloring the graph with a minimal number  $k$  of colors we compute the set  $P = \{\{T_j | T_j \text{ has color } i\} | 1 \leq i \leq k\}$ .  $P$  is a minimal disjoint sets partitioning.
- Reduce MGC to mDSPP:  
Let  $G = (V, E)$  be an undirected graph. We get the corresponding mDSPP by computing for each node  $v \in V$  the set  $T_v = \{\{v, w\} \in E\}$  and the set  $T = \{T_v | v \in V\}$ . Then we solve the mDSPP for this  $T$ . Let  $P = \{P_1, \dots, P_k\}$  be the resulting partitioning, then for every  $T_v \in P_i$  the node  $v$  gets color  $i$ .

□

**Theorem 1**

mGAPP is  $\mathcal{NP}$ -complete.

*Proof:*

- Reduce mGAPP to mDSPP:  
A node in a GAP represents several nodes with disjoint live times. In a minimal GAP nodes are grouped together, such that there are a minimal number of groups and the live times of the nodes in a group are disjoint.

Thus the mGAPP computation is reduced to solving a mDSPP for their live times  $T_1 = T(v_1), \dots, T_n = T(v_n)$ .

- Reduce mDSPP to mGAPP:

It remains to reduce a mDSPP  $T = \{T_1, \dots, T_n\}$  to a mGAPP. For this we compute the following graph animation  $G = [g_{t_1}, \dots, g_{t_m}]$  with  $t_i \in \mathcal{T} = \bigcup_{i=1}^n T_i$ ,  $g_{t_i} = (\{T_j | t_i \in T_j\}, \emptyset)$  and  $m = |\mathcal{T}|$ . Let  $\tilde{g} = (\{P_1, \dots, P_k\}, \emptyset)$  a minimal GAP of  $G$ , then  $P = \{P_1, \dots, P_k\}$  is the solution of the mDSPP.

□

Now we present an algorithm which computes a GAP in  $O(n^2)$  where  $n$  is the number of nodes of the super graph.

**Algorithm 1 (Computing a GAP)**

$W := \widehat{V}, P := [], p := 0$

While  $v \in W$  do

  If  $\exists j : T(v) \cap T(P_j) = \emptyset$  then

$P_j := P_j \cup \{v\}, T(P_j) := T(P_j) \cup T(v)$

  else

$p := p + 1, P_p := \{v\}, T(P_p) := T(v)$

$W := W - \{v\}$

**Theorem 2 (Total Correctness of Algorithm 1)**

The algorithm terminates and the graph  $(\bigcup_{i=1}^p P_i, \tilde{E})$ , where  $\tilde{E}$  is the induced set of edges, is a GAP.

To prove the above theorem we prove the following stronger lemma:

**Lemma 2**

After and before each iteration  $(\bigcup_{v \in W} \{v\} \cup \bigcup_{i=1}^p P_i, \tilde{E})$  is a GAP.

*Proof:*

**Induction basis:** Before the first iteration  $W = \widehat{V}$  and  $P = []$  and thus the lemma holds.

**Induction step:** Assume that the lemma holds before the next iteration and we select  $v^* \in W$ . There are two cases:

**first case:** Test  $T(v^*) \cap T(P_j) = \emptyset$  is true.

Then we get a new graph  $(\bigcup_{v \in W, v \neq v^*} \{v\} \cup \bigcup_{i=1, i \neq j}^p P_i \cup \{P_j \cup \{v^*\}\}, \tilde{E})$ . Only the node  $P_j$  got an additional element  $v^*$ . Because of the above test, it satisfies the condition of Definition 6.

**second case:** The added node  $\{v^*\}$  is a singleton and thus satisfies the condition of Definition 6.

Because after each iteration  $W$  is reduced by one, the algorithm terminates. At the end of the algorithm  $W$  is empty and from the lemma follows Theorem 2.  $\square$

### 3.2 Strategies for Computing a GAP

From an aesthetical point of view it is not too bad that we do not compute minimal GAP's. A minimal GAP is often not the best choice as we pay for the minimal number of nodes by an increased number of edge crossings. In Algorithm 1 we have not specified in which order the life times of the node  $v$  and the already computed partitions  $P_j$  are compared, i.e. how to find a  $j$  such that  $T(v) \cap T(P_j) = \emptyset$ . In our implementation we can choose one of the following strategies which in general yield different GAPs:

1. Search the list from  $P_1$  to  $P_p$ .
2. Search the list from  $P_p$  to  $P_1$ .
3. Add  $v$  to the partition with the smallest number of nodes.
4. Only allow a limited number of nodes in a partition. If there is no partition with less nodes, then create a new partition.
5. Only allow a limited number of edges in a partition.
6. Give priority to nodes with induced edges to the same already computed partitions.

### 3.3 Reduced Graph Animation Partitionings

In a GAP the number of nodes of the super graph of a graph animation is reduced. In a similar way, the number of edges can be reduced.

**Definition 8 (Reduced Graph Animation Partitioning RGAP)**

Let  $G = [g_1, \dots, g_n]$  with  $g_i = (V_i, E_i)$  be a graph animation and  $\tilde{g} = (\tilde{V}, \tilde{E})$  be a GAP of  $G$ . The graph  $\tilde{g} = (\tilde{V}, \tilde{E})$ , where  $\tilde{E} \subseteq \tilde{V} \times \tilde{V} \times \mathcal{P}(\text{Id})$ , is a **reduced GAP**, iff  $\forall (\tilde{v}_1, \tilde{v}_2, \{m_1, \dots, m_k\}) \in \tilde{E}$  the following holds:

$$(\tilde{v}_1, \tilde{v}_2, m_i), (\tilde{v}_1, \tilde{v}_2, m_j) \in \tilde{E} : T(m_i) \cap T(m_j) = \emptyset \text{ for } 1 \leq i < j \leq k$$

We call  $\tilde{g}$  a *minimal RGAP* of  $G$ , if there exists no RGAP of  $G$  with less edges.

An edge  $(\tilde{v}_1, \tilde{v}_2, \{m_1, \dots, m_k\})$  of the RGAP represents  $k$  edges which exist at different times, i.e. in different graphs of the graph animation, between a node in  $\tilde{v}_1$  and  $\tilde{v}_2$ . But it does not represent two or more multi-edges which exist at the same time; they can not be represented by a single edge in the RGAP. Also the problem of computing a minimal RGAP (hence mRGAPP) is  $\mathcal{NP}$ -complete.



**Theorem 3**

*mRGAPP is  $\mathcal{NP}$ -complete.*

*Proof:*

- Reduce mRGAPP to mDSPP:

An edge in an RGAP represents several edges with disjoint live times. Consider two nodes  $\tilde{v}_1$  and  $\tilde{v}_2$  and let  $\{m_1, \dots, m_n\}$  be the edges between these two nodes in the GAP. In a minimal RGAP edges are grouped together, such that there are a minimal number of groups and the live times of the edges in a group are disjoint. Thus for each pair of nodes a mDSPP is solved for their live times  $T_1 = T(m_1), \dots, T_n = T(m_n)$ .

- Reduce mDSPP to mRAGPP:

It remains to reduce a mDSPP  $T = \{T_1, \dots, T_n\}$  to a mRGAPP. For this we compute the following graph animation  $G = [g_{t_1}, \dots, g_{t_m}]$  with  $t_i \in \mathcal{T} = \bigcup_{i=1}^n T_i$ ,  $g_{t_i} = (\{v_1, v_2\}, \{(v_1, v_2, T_j) | t_i \in T_j\})$  and  $m = |\mathcal{T}|$ . Let  $\tilde{g} = (\{\{v_1\}, \{v_2\}\}, \{(v_1, v_2, P_1), \dots, (v_1, v_2, P_k)\})$  be a minimal RGAP of  $G$ . Then  $P = \{P_1, \dots, P_k\}$  is a solution of the mDSPP. □

As computing minimal RGAPs is  $\mathcal{NP}$ -complete, we present a faster algorithm ( $O(m^2)$  where  $m = |\widehat{E}|$ ) which does not compute minimal RGAPs, but yields good results in practice, i.e. RGAPs with small numbers of edges. The algorithm actually computes the partitioning of the edge identifiers for an RGAP.

**Algorithm 2 (Computing a RGAP)**

$W := \{m_1, \dots, m_k\}$ , i.e. the set of all identifiers occurring in  $\tilde{E}$

$P := [], p := 0$

While  $n \in W$  do

Let  $(\tilde{v}, \tilde{w}, n)$  be the edge identified by  $n$ .

$p := p + 1, P_p := \{n\}, T(P_p) := T(n)$

While  $\exists m \in W$  with  $(\tilde{v}, \tilde{w}, m)$  and  $T(P_p) \cap T(m) = \emptyset$  then

$P_p := P_p \cup \{m\}, T(P_p) := T(P_p) \cup T(m), W := W - \{m\}$

$W := W - \{n\}$

**Theorem 4 (Total Correctness of Algorithm 2)**

*The algorithm terminates and the graph  $(\tilde{V}, \{(\tilde{v}, \tilde{w}, P_i) | 1 \leq i \leq p \text{ and } \exists(\tilde{v}, \tilde{w}, n) \in \tilde{E} \text{ and } n \in P_i\})$  is an RGAP.*

To prove the above theorem we prove the following lemma:

**Lemma 3**

*After and before each iteration of the outer loop of Algorithm 2 the graph  $(\tilde{V}, \{(\tilde{v}, \tilde{w}, P_i) | \exists(\tilde{v}, \tilde{w}, n) \in \tilde{E} \text{ and } n \in P_i\} \cup \{(\tilde{v}, \tilde{w}, \{n\}) | n \in W \text{ and } (\tilde{v}, \tilde{w}, n) \in \tilde{E}\})$  is an RGAP.*

*Proof:*

**Induction basis:** Before the first iteration  $W = \{m_1, \dots, m_n\}$  and  $P = []$ . This means that there is no edge  $(\tilde{v}_1, \tilde{v}_2, \{m_1, \dots, m_k\})$  with  $k > 1$  in the RGAP and thus the lemma holds.

**Induction step:** Assume that the lemma holds before the next iteration of the outer loop and we select  $n \in W$ . Then we build a new singleton partition  $\{n\}$  and the lemma still holds. In the inner loop we add edges to the current partition which have disjoint live times with all edges that are already in the partition, which means that all edges in one partition are in different  $E_q$  and thus the lemma also holds after the inner loop.

Because in each loop  $W$  is reduced by one, the algorithm terminates. At the end of the algorithm  $W$  is empty and from the lemma follows Theorem 4.  $\square$

### 3.4 Algorithm

After we have seen how to compute RGAPs, we now show how they can be used in combination with a static graph layouter to draw a sequence of graphs while preserving the mental map.

**Algorithm 3 (Foresighted Layout)**

```
foresightedLayout( $[g_1, \dots, g_k]$ , staticLayouter()) {  
     $g = \text{computeGAP}(g_1, \dots, g_k)$   
     $\bar{g} = \text{computeRGAP}(g)$   
    layout = staticLayouter( $\bar{g}$ )  
    for  $i = 1$  to  $k$   
        drawGraph( $g_i$ , layout)  
}
```

We call the static layouter to compute a layout of the RGAP of the graph animation. We assume that the static layouter returns a layout, i.e. a data structure containing the positions of each node and polylines (or bends) for each edge. The function `drawGraph()` gets this data structure and a graph of the graph animation. For each node in the graph it uses the layout information of its super node, i.e. the node in the RGAP it is a member of. For each edge it uses the layout information of the bends of the edge in the RGAP which contains its identifier.

## 4 Implementation

We have implemented Foresighted Layout in Java as part of an API which we use for algorithm animations. The class `AnimatedGraph` of this API has the following interface:

```

class AnimatedGraph {
    public AnimatedGraph(GView view)
    public void insertNode(Node n)
    public void insertEdge(Edge e)
    public void deleteNode(Node n)
    public void deleteEdge(Edge e)

    public void snapshot()

    public void play()
    public void next()
    public void back()

    public void perform(Object target, String methodname, Object arg)
        throws NoSuchMethodException
    public void perform(Object target, String methodname, Object arg,
        Object reverseTarget, String reverseMethodname,
        Object reverseArg)
        throws NoSuchMethodException
}

```

The class provides methods to build and modify a graph, to record a graph animation by doing snapshots of individual graphs and replay the animation afterwards.

A node can be a specialization of any AWT component which has to implement a certain interface (a few additional methods). Thus it is also possible to draw a graph in a node of a graph again. As the nodes can be AWT components, one can also destructively change attributes of these objects during the recording sessions. To defer these changes until the animation is replayed, such changes must be done using the method `perform()`, which puts the method calls into a data structure and invokes them later using Java Reflection.

## 5 Examples

The basic idea of the static layout algorithm used in our examples is to divide the nodes into several levels. Then the algorithm computes the relative positions of the nodes within these levels, so that edge crossings are minimized [10, 8]. Ideally this method can be used for directed graphs, because the direction of the edges can be used for the layouting process.

### 5.1 Algorithm Animation

The GaniFA applet visualizes and animates several generation algorithms from automata theory including the generation of a non-deterministic finite automaton

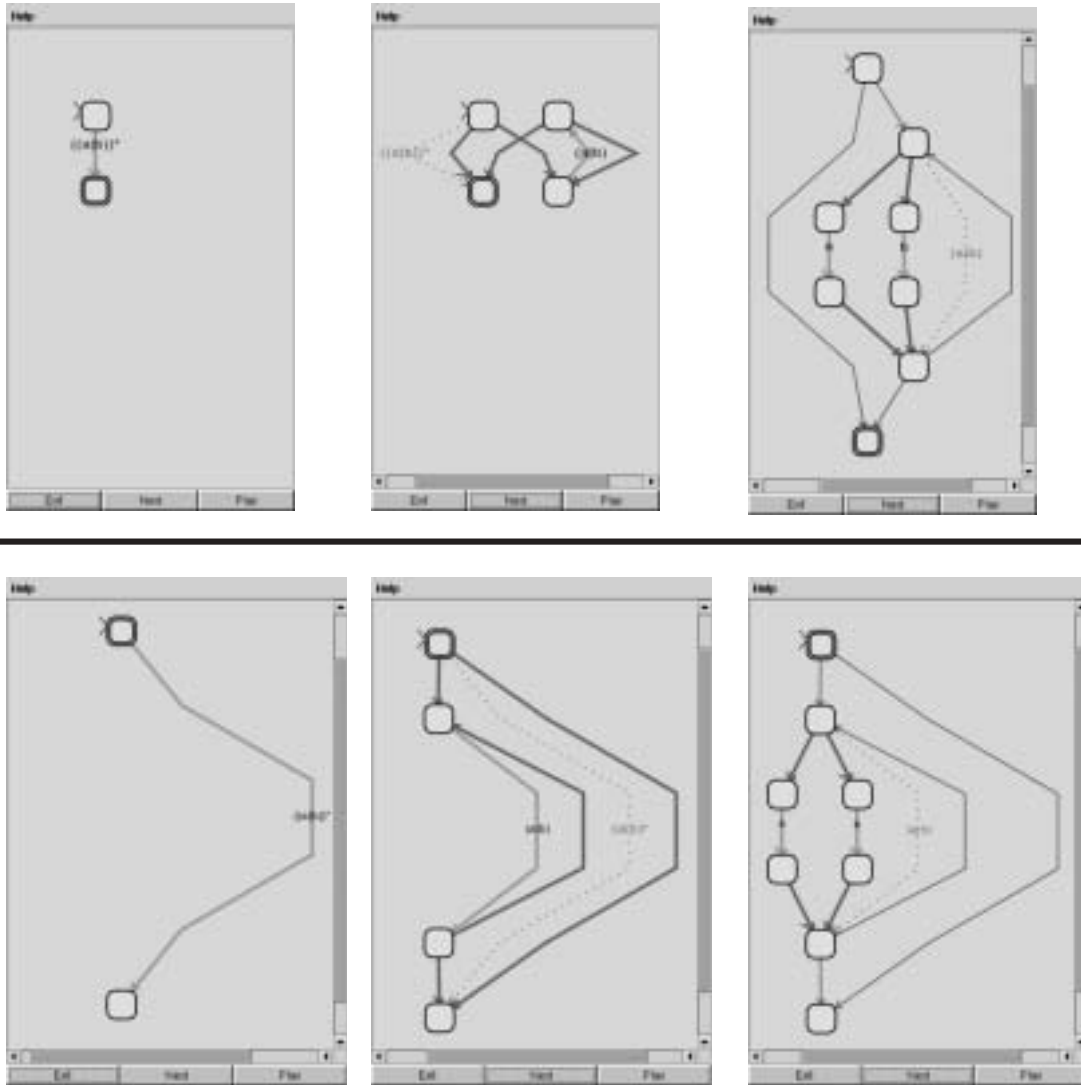


Figure 1: Ad-hoc and Foresighted Layout of the intermediate and final NFA for  $(a|b)^*$ .

(NFA) from a regular expression RE [11]. We have included GaniFA into an electronic textbook on automata theory to allow interactive exercises [2, 3, 4].

In case of visualizing transition diagrams of finite automata our static layout algorithm is a good choice, but the algorithm  $RE \rightarrow NFA$  changes the graph successively.

Animations of algorithms which change graphs, i.e. add or delete nodes and edges, are often very confusing, because after each change a new layout of the current graph is computed. In this new layout nodes are moved to different places although the algorithm didn't actually change these nodes. As a result it is not clear to the user what changes of the graph are due to the graph algorithm and

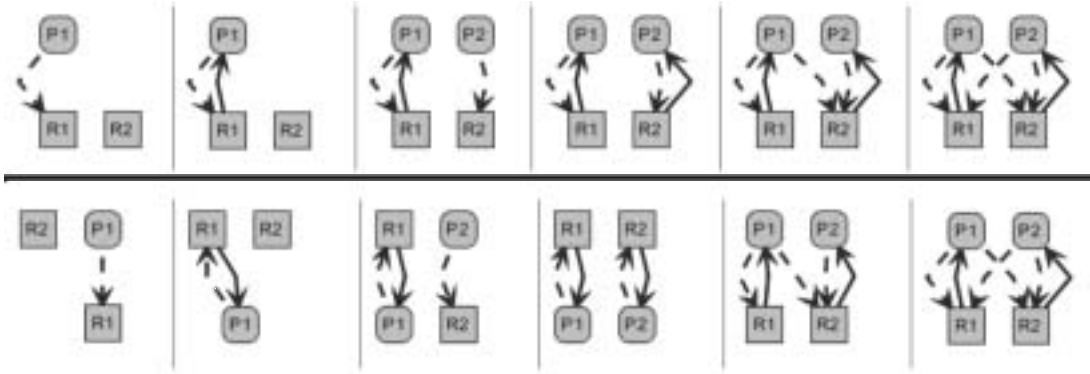


Figure 2: Foresighted and Ad-Hoc Layout for deadlock situation.

what changes are due to the layout algorithm.

The lower part of Figure 1 shows how Foresighted Layout can be used to animate the conversion of a regular expression  $(a|b)^*$  into an appropriate non-deterministic finite state automaton ( $RE \rightarrow NFA$ ). In contrast to the upper part of Figure 1, which shows the same conversion, this visualization is significantly more clear because once created, a node doesn't change its position.

## 5.2 Resource Allocation

In Figure 2 we show a classical deadlock situation which in practice can be avoided by using ordered resources. Process P1 requests resource R1 (indicated by the dashed arrow) and gets exclusive access (indicated by the solid arrow). Then P2 requests and gets exclusive access to R2. Next P1 requests access to R2, but the resource is locked by P2. Then P2 requests access to R1 which is locked by P1. As long as none of the two processes releases its lock both are stuck.

In the animation using ad-hoc layout it is difficult to see what is changed between subsequent graphs. In the 6 pictures of the ad-hoc layout P1 is drawn at 4, P2 at 2, R1 at 3 and R2 at 4 different positions. As a result for example the dashed arrow from P1 to R1 is drawn upwards in some and downwards in other pictures. Using Foresighted Layout nodes remain at their position and edges are drawn consistently in all graphs.

## 5.3 Buffered I/O

In Figure 3 we show four different techniques to layout a graph animation. In this example several users share a printer, but the access to the printer is buffered by using a printer spool. In the ad-hoc layout the positions of the nodes "Printer" and "Spool" change several times. In the supergraph-based foresighted layout node positions are fixed, but there is much unused space. In the GAP-based foresighted layout node positions do also not change, but as the nodes "User1"

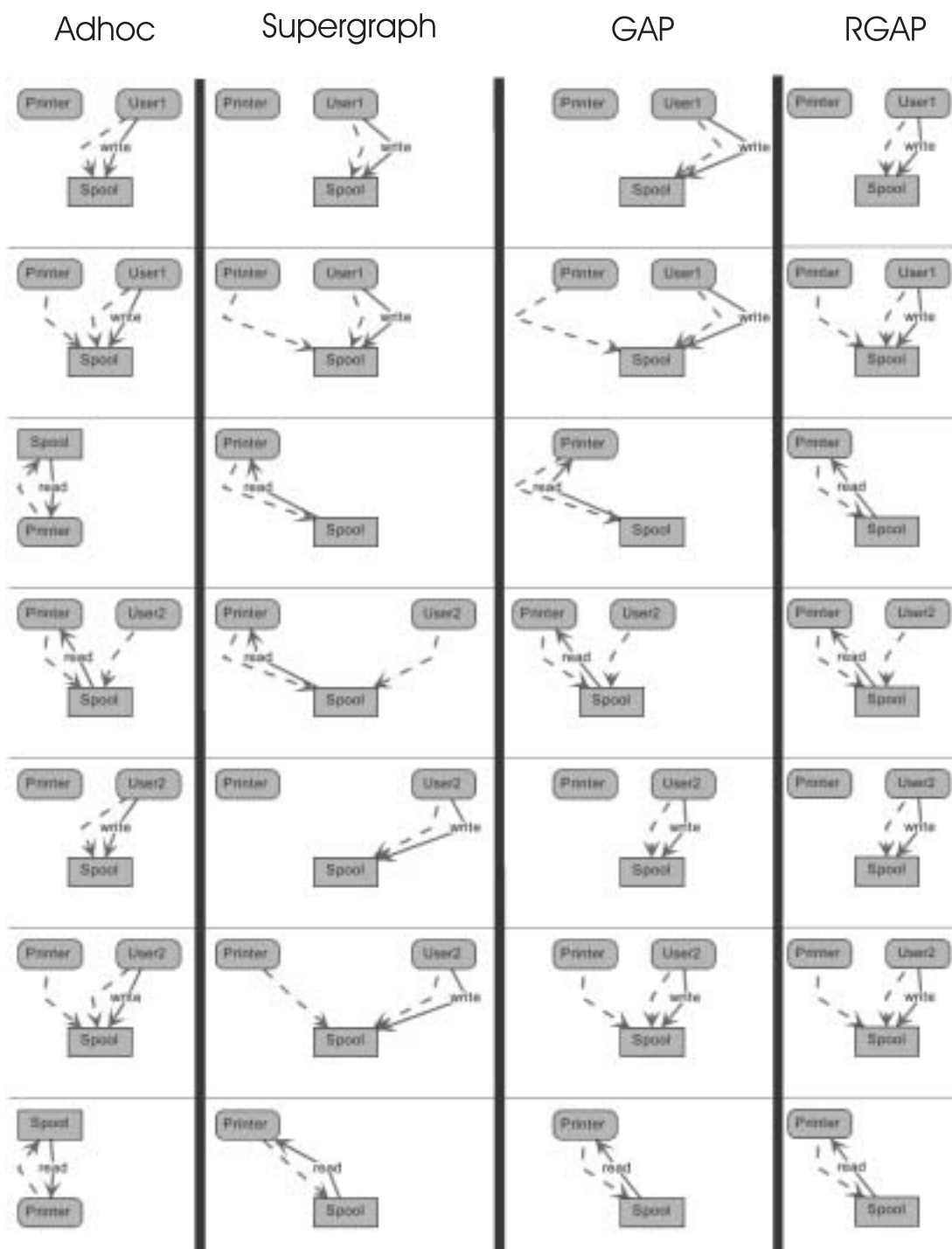


Figure 3: Different graph animations of buffered I/O example.

and "User2" share the same position, the layout is more compact. But there are sharp bends in the edges of the first three graphs and normal bends in those of the last four graphs. Finally in the RGAP-based foresighted layout there are no sharp bends as edges at different live times share bend positions. Obviously, for this example RGAP-based foresighted layout produces the best results.

## 6 Interactive Graph Animations

In most applications the future of a graph depends on user input. Nevertheless between such points in time when the user interacts with the application, the program can perform several "foreseeable" changes of the graph. Thus the execution of such an interactive application can be modeled as a sequence of graph animations. When we draw a graph animation of such a sequence on the screen, we do not know the next animation in the sequence but we know the one before. As a "smooth" transition between the previous and the actual graph animation we can use the traditional morphing approach. More precisely: Let  $G = [g_1, \dots, g_n]$  be the previously drawn graph animation. Then graph  $g_n$  was drawn on the screen using the Foresighted Layout for an RGAP  $\bar{g}$  of  $G$ . Now the user does some input and triggers the graph animation  $G' = [g'_1, \dots, g'_k]$ . To draw this animation the application computes an RGAP  $\bar{g}'$  of  $G'$  and uses morphing between the graph  $g_n$  with node and edge positions as in  $\bar{g}$  and  $g'_1$  with node and edge positions as in  $\bar{g}'$ .

## 7 Conclusion

We have presented the motivation and theory behind Foresighted Layout. Using our generic algorithm existing static graph drawing algorithms can be used for graph animations which preserve the mental map. The algorithm has been implemented in Java and in particular used for algorithm animations. For this kind of application it provides better results than traditional approaches which use smooth transitions and/or incremental changes of the layout, e.g. using the VCG tool [9].

**Acknowledgement** This research has been partially supported by the German Research Council (DFG) under grant WI 576/8-1 and WI 576/8-3.

## References

- [1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.
- [2] S. Diehl and A. Kerren. Increasing explorativity by generation. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000*. AACE, 2000.
- [3] S. Diehl and A. Kerren. Levels of exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001 (to appear)*, 2001.
- [4] GANIMAL. Project homepage. <http://www.cs.uni-sb.de/GANIMAL>.
- [5] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.
- [6] I. Herman, G. Melancon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.
- [7] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.
- [8] G. Sander. *Visualization Techniques for Compiler Construction*. Dissertation (in german), University of Saarland, Saarbrücken (Germany), 1996.
- [9] G. Sander, M. Alt, C. Ferdinand, and R. Wilhelm. Clax - a visualized compiler. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Computer Science*. Springer-Verlag, 1996.
- [10] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.
- [11] Reinhard Wilhelm and Dieter Maurer. *Compiler Design: Theory, Construction, Generation*. Addison-Wesley, 2nd printing edition, 1996.