Preserving the Mental Map using Foresighted Layout

Stephan Diehl, Carsten Görg and Andreas Kerren

University of Saarland, FR 6.2 Informatik,
PO Box 15 11 50, D-66041 Saarbrücken, Germany
{diehl,goerg,kerren}@cs.uni-sb.de

Abstract. First 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, trees can be animated with a static tree layouter, graphs with a static Sugiyama-style layouter or a spring embedder, etc. Second we discuss applications of Foresighted Layout in algorithm animation and visualization of navigation behaviour.

1 Introduction

Most work on graph drawing addresses the problem of layouting 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, 7]. 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 layouting 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" [8].

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 accomodate the update. Unfortunately, in the worst case they have to compute the layout of the whole graph.

In this paper 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 features 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 accepts 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.

In particular applications can be visualized post mortem using information stored in log files. In Sections 5 and 6 we discuss two applications of Foresighted Layout. First in the area of algorithm animation we used it to visualize the generation of finite state automata which are drawn as state transition diagrams. Second, in the area of web visualization, we have implemented a portal which logs the web pages and links visited by a user. We visualize these logs later to analyse the navigation behavior of different users.

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* $E \subseteq V \times V \times \mathrm{Id}$ *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* $G = [g_1, \ldots, g_n]$ *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* $G = [g_1, \ldots, g_n]$ *with* $g_i = (V_i, E_i)$*, then the* **super graph** $\widehat{G}$ *of* G *is defined as* $\widehat{G} = (\widehat{V}, \widehat{E})$ *with* $\widehat{V} = \bigcup_{i=1}^{n} V_i$ *and* $\widehat{E} = \bigcup_{i=1}^{n} E_i$.

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 $G = [g_1, \ldots, g_n]$ be a graph animation and $\widehat{G} = (\widehat{V}, \widehat{E})$ 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 $v \in \widehat{V}$ 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 $\widetilde{V} \subseteq \mathcal{P}(V)$ and $\widetilde{E} \subseteq \widetilde{V} \times \widetilde{V} \times \mathrm{Id}$. A graph $\widetilde{g} = (\widetilde{V}, \widetilde{E})$ is a* **graph partitioning** *of g iff the nodes in $\widetilde{V}$ are disjoint, $\bigcup_{v \in \widetilde{V}} v = V$ and $(\widetilde{v_1}, \widetilde{v_2}, n) \in \widetilde{E} \Leftrightarrow \exists v_1 \in \widetilde{v_1}$ and $v_2 \in \widetilde{v_2} : (v_1, v_2, n) \in E$. We call $\widetilde{E}$ the set of edges induced by $\widetilde{V}$.*

In other words, $\widetilde{V}$ is a partitioning of V. Each node in $\widetilde{V}$ 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 $G = [g_1, \ldots, g_n]$ 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 $\widetilde{g} = (\widetilde{V}, \widetilde{E})$ of $\widehat{G}$ where $\widetilde{V} = \{P_1, \ldots, P_k\}$ 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. We have proven the $\mathcal{NP}$-completeness of mGAPP and mRGAPP (see Section 3.3) by reduction on the minimal graph coloring problem [2, 6]. 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
    $p := p + 1, P_p := \{v\}, T(P_p) := T(v)$
  $W := W - \{v\}$

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 7 (Reduced Graph Animation Partitioning RGAP). *Let $G = [g_1, \ldots, g_n]$ with $g_i = (V_i, E_i)$ be a graph animation and $\widetilde{g} = (\widetilde{V}, \widetilde{E})$ be a GAP of G. The graph $\overline{g} = (\widetilde{V}, \overline{E})$, where $\overline{E} \subseteq \widetilde{V} \times \widetilde{V} \times \mathcal{P}(\mathrm{Id})$, is a* **reduced GAP**, *iff $\forall (\widetilde{v_1}, \widetilde{v_2}, \{m_1, \ldots, m_k\}) \in \overline{E}$ the following holds:*
$(\widetilde{v_1}, \widetilde{v_2}, m_i), (\widetilde{v_1}, \widetilde{v_2}, m_j) \in \widetilde{E} : T(m_i) \cap T(m_j) = \emptyset$ for $1 \leq i < j \leq k$
We call $\overline{g}$ a minimal RGAP of G, if there exists no RGAP of G with less edges.

An edge $(\widetilde{v_1}, \widetilde{v_2}, \{m_1, \ldots, m_k\})$ of the RGAP represents k edges which exist at different times, i.e. in different graphs of the graph animation, between a node in $\widetilde{v_1}$ and $\widetilde{v_2}$. 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.

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).
$W := \{m_1, \ldots, m_k\}$, i.e. the set of all identifiers occuring in $\widetilde{E}$
$P := [\,], p := 0$
While $n \in W$ do
  Let $(\widetilde{v}, \widetilde{w}, n)$ be the edge identified by n.
  $p := p + 1, P_p := \{n\}, T(P_p) := T(n)$

While $\exists m \in W$ with $(\widetilde{v}, \widetilde{w}, m)$ and $T(P_p) \cap T(m) = \emptyset$ then
$P_p := P_p \cup \{m\}, T(P_p) := T(P_p) \cup T(m), W := W - \{m\}$
$W := W - \{n\}$

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 (Forsighted Layout).

```
foresightedLayout([g_1, ..., g_k], staticLayouter())
 {  g =computeGAP(g_1, ..., g_k)
    g̅ =computeRGAP(g)
    layout=staticLayouter(g̅)
    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 [5]. 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.

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 [11, 9]. Ideally this method can be used for directed graphs, because the direction of the edges can be used for the layouting process.

4.1 Drawing Graphs in 4 Dimensions

In addition to showing the different graphs in a graph animation one after another in 2D, we have implemented a 3D viewer in Java3D which uses the third dimension as a time axis. As a result we can show several graphs simultaneously in a history view. In addition the 3D viewer can show the supergraph in the background, see Figure 3. Finally it allows to interact and customize the view in various ways including recolouring, translating and rotating the graph.

5 Algorithm Animation

Algorithm animation is one of the most prominent areas of software visualization. The GaniFA applet visualizes and animates several generation algorithms from automata theory including the generation of a non-deterministic finite automaton (NFA) from a regular expression RE [12]. We have included GaniFA into an electronic textbook on automata theory to allow interactive exercises [3–5].

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

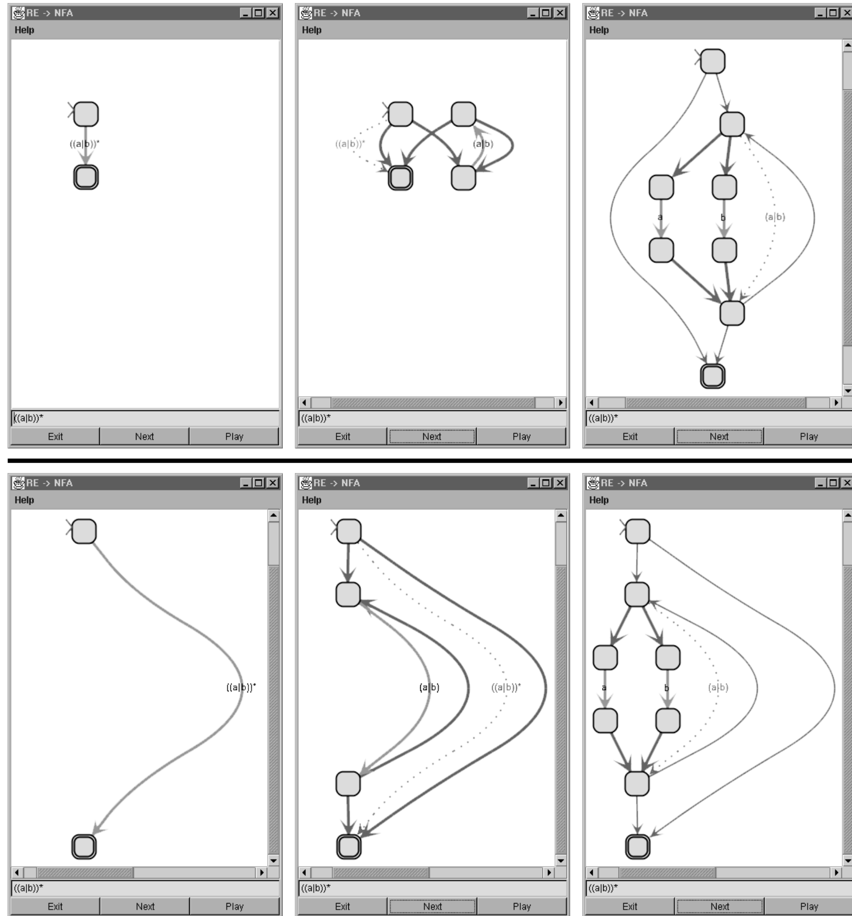

Fig. 1. Ad-hoc and foresighted layout of the intermediate and final NFA for $(a|b)^*$.

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 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 nondeterministic 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.

6 Visualization of Navigation Behavior

As a user browses through web pages he unfolds a subgraph of the web. As he moves from one page to another, new pages become directly accessible as hy-

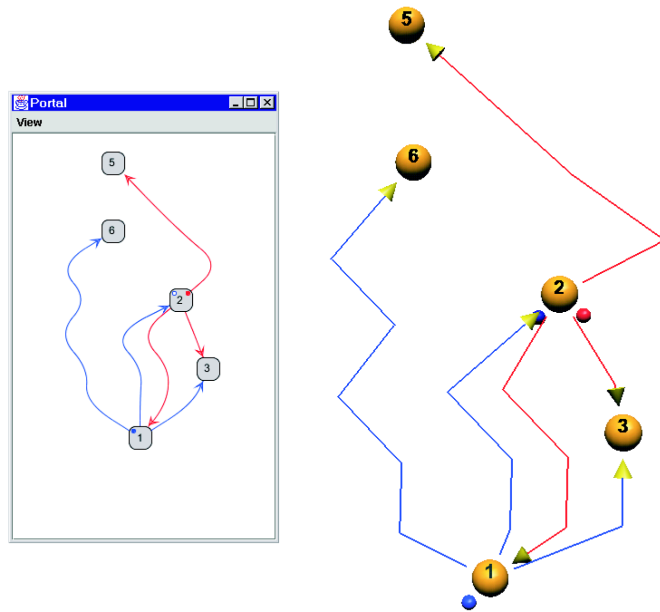

Fig. 2. A graph in 2D and 3D view.

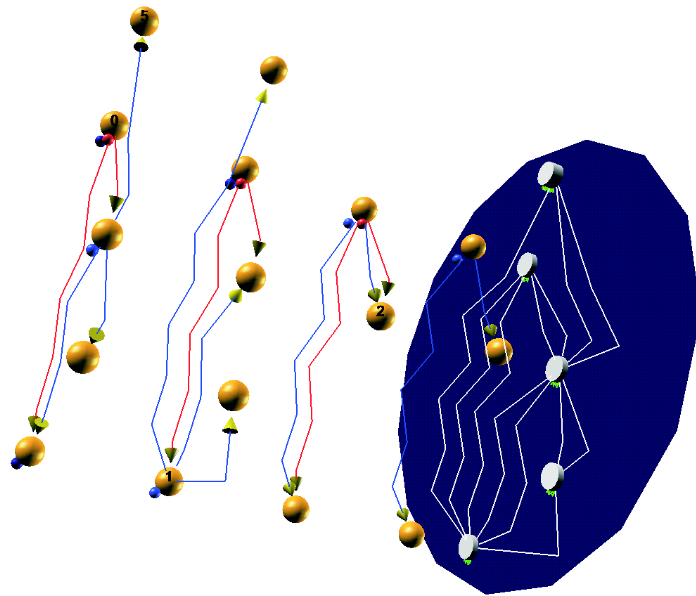

Fig. 3. A history view of navigation by two users.

perlinks, whereas those which have been directly accessible before, might not be accessible from the current page. To analyse the browsing or navigation behavior of one or several users we visualize the subgraphs which they are currently aware of. We can visualize the subgraphs of different users at the same time sharing nodes, if these represent the same web page. The edges in each subgraph are drawn in the users color. Thus we see when they visit the same page or have a links to the same pages. In Figure 2 we see, that both users are currently on web page 2 as indicated by the two circles with the colors of the users in node 2.

There are various ways to acquire the necessary information in log files. We have implemented a web page, which we call portal, which logs the web pages and links visited by a user who enters our web server through this portal. We visualize these logs later to analyse the navigation behavior of different users.

With the help of the portal and a tool which uses foresighted layout it is possible to analyse the learner navigation through the electronic textbook discussed above. Thus, we can see what pages a student looks at or that a student frequently uses the glossary. If we visualize the behavior of several students we might find that different students catch up (synchronize) on certain pages, although their navigation behavior differs considerably on intermediate pages.

These analyses are facilitated by the history view of the 3D version of our foresighted graphlayout, see Figure 3.

7 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 $G = [g_1, \ldots, g_n]$ be the previously drawn graph animation. Then graph g_n was drawn on the screen using the Foresighted Layout for an RGAP $\overline{g}$ of G. Now the user does some input and triggers the graph animation $G' = [g'_1, \ldots, g'_k]$. To draw this animation the application computes an RGAP $\overline{g'}$ of G' and uses morphing between the graph g_n with node and edge positions as in $\overline{g'}$ and g'_1 with node and edge positions as in $\overline{g'}$.

8 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 and visualization of navigation behaviour. For these kinds 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 [10]. For the analysis of navigation behaviour the history of our 3D viewer turned out to be very beneficial.

Acknowledgement This research has been partially supported by the German Research Council (DFG) under grant WI 576/8-1 and WI 576/8-3. For helping with the implementation of the 3D viewer the authors thank Peter Blanchebarbe.

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, C. Görg, and A. Kerren. Foresighted Graphlayout. Technical Report A/02/2000, FR 6.2 - Informatik, University of Saarland, December 2000. http://www.cs.uni-sb.de/tr/FB14.
3. S. Diehl and A. Kerren. Increasing Explorativity by Generation. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000*. AACE, 2000.
4. S. Diehl and A. Kerren. Levels of Exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001*. ACM, 2001.
5. GANIMAL. Project Homepage. http://www.cs.uni-sb.de/GANIMAL.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. Freeman and Company, 1979.
7. 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.
8. 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.
9. G. Sander. *Visualization Techniques for Compiler Construction*. Dissertation (in german), University of Saarland, Saarbrücken (Germany), 1996.
10. 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.
11. 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.
12. Reinhard Wilhelm and Dieter Maurer. *Compiler Design: Theory, Construction, Generation*. Addison-Wesley, 2nd printing edition, 1996.