Graph Layout Through the VCG Tool

(Extended Abstract and System Demonstration*)

Georg Sander** (sander@cs.uni-sb.de)
Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken

Abstract. The VCG tool allows to visualize graphs that occur typically as data structures in programs. We describe the functionality of the VCG tool, its layout algorithm and its heuristics. Our main emphasis in the selection of methods is to achieve a very good performance for the layout of large graphs. The tool supports the partitioning of edges and nodes into edge classes and nested subgraphs, the folding of regions, and the management of priorities of edges. The algorithm produces good drawings and runs reasonably fast even on very large graphs.

1 Introduction

Visualization allows better understanding of the behavior of data structures in programs. Especially in compilers – as they are developed by the ESPRIT project #5399 COMPARE (Compiler Generation for Parallel Architectures) [1] – many parts of the data structures are trees or graphs, e.g., the syntax tree, the control flow graph, the call graph or the data dependence graph [15]. A simple textual visualization of them is too confusing or even unreadable. A special visualization tool that draws them in a natural way is more helpful.

Since the calculation of a nice layout is computationally hard [4] and data structure representations are often very large, an interactive graph drawing tool must use heuristics and needs facilities to reduce the amount of information to be displayed. Furthermore, data structures are often interwoven, and parts of them are more important than others in a certain situation, such that it must be possible to assign priorities to the parts of the graph whose structure must be more readable.

For graph drawing, there are some common aesthetic criteria like avoiding crossings of edges and nodes, favoring short straight edges and balancement. An *interactive* tool has the additional, important criterion: *Be reasonably fast.* Thus, the main emphasis was to select methods and heuristics to be able to deal fast with large graphs.

In the following sections, we sketch the layout method of the VCG tool. This method has common parts with the layout algorithms of similar tools (see [2][12][8]), but in many cases, it is faster or can deal with larger graphs. Our work is based on the approach of Sugiyama, Tagawa, and Toda [13]. We extend their algorithm by several new heuristics, by avoiding their expensive matrix

* The full article is available as Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 1994, and via ftp from ftp.cs.uni-sb.de (134.96.7.254), file /pub/graphics/vcg/doc/tr-A03-94.ps.gz.

** This work is supported by the ESPRIT Project #5399 COMPARE

operations in order to be able to handle large graphs, and by the possibility to include priorities and anchor points of edges in the layout algorithm.

2 Problem Description

In this section, we introduce some general graph notions and explain the task of the layout algorithm. The graph is given to the VCG tool by a textual specification of nested subgraphs, nodes and edges, annotated by attributes:

Definition 1 *A* **nested graph** *$G = (V, E)$ consists of a set V of nodes and a set $E \subseteq V \times V$ of edges. A node $v \in V$ is either a simple node or a nested graph G. If a graph does not contain nested subgraphs, we call the graph* **flat**.
Edges have the following attributes: an anchor point, a priority and a class.

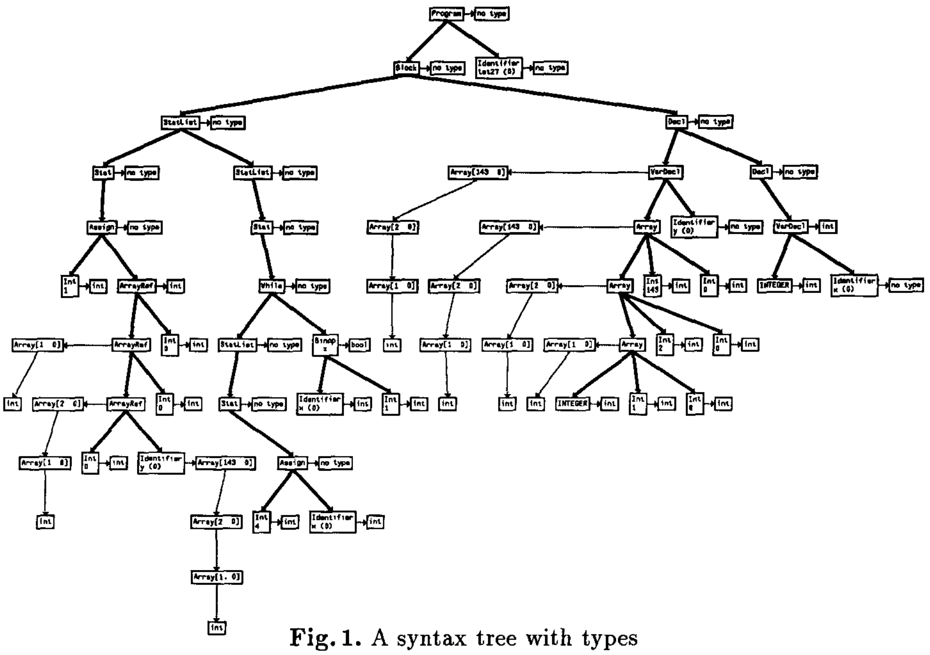
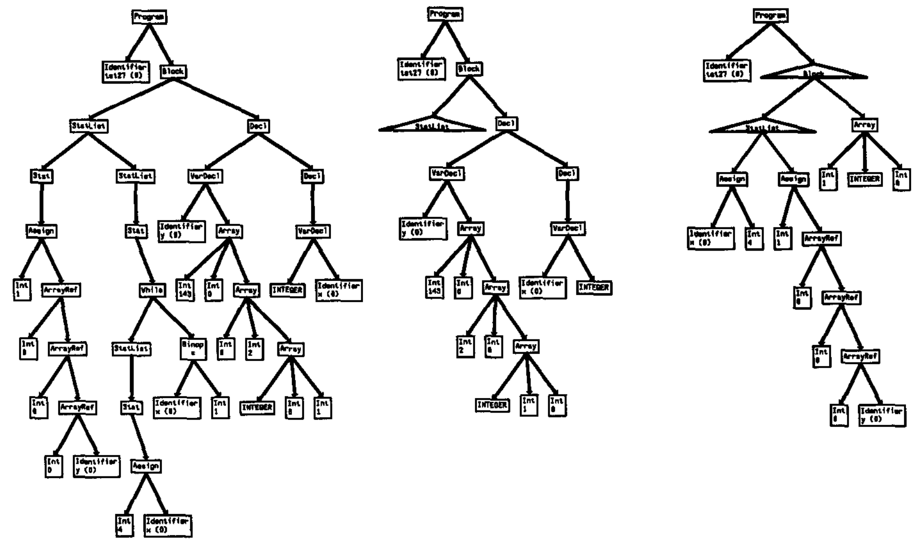


Fig. 1. A syntax tree with types

The attributes can be specified by the user and allow to influence the appearance of the graph. Graphs can be folded in different ways. The aim of a folding operation is to reduce the amount of the objects that have to be laid out. This allows the user to select interactively parts that must be inspected and to hide parts that currently are not of interest:

- **subgraph folding**: A subgraph is normally laid out by displaying all its nodes. But it can also be folded, i.e. all its nodes and edges disappear. Instead, a summary node is drawn.

- **edge class hiding**: All edges of an edge class can be hidden. They disappear. Single nodes that are not anymore connected with the rest of the graph are removed, too.
- **region folding**: Given a set of start nodes, a set of end nodes and a predicate P on edge classes, the corresponding region consists of all nodes that are reachable from a start node by a path that does not contain an end node, where P is true for all edges of the path. The region can be folded into one summary node.



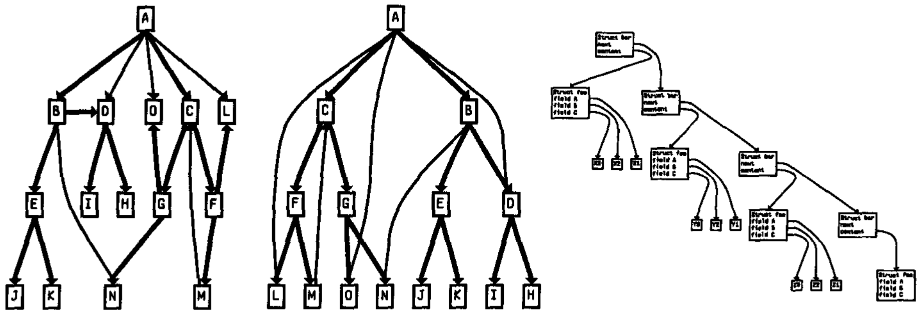
Annotations are hidden A subtree is folded Regions are folded

Fig. 2. The same syntax tree: Foldings

Figure 1 shows a syntax tree annotated by type information. The syntax tree edges have the class 1, while the annotations are connected by edges of class 2. The result of the folding operation 'Hide Edge Class 2' is shown in fig. 2 (left): The type information has disappeared. In fig. 2 (middle), the region starting at the uppermost 'StatList' node is folded and represented by a triangle. In fig. 2 (right), the region starts with the same node but end at the both 'Assign' nodes. A second region starts at the 'Block' node and ends at this folded 'StatList' node and at the lowermost 'Array' node. In our example, we used the predicate 'e has edge class ≤ 2' for these operations 'Fold Region'.

The priority p indicates the importance of the edge. Edges of high priority are preferred, thus they are shorter and have more influence on the structure of the layout (fig. 3 left and middle).

The anchor point of an edge specifies the point where the edge is anchored at the source node. The visualization of data structures containing several fields of pointers often require the association of an edge with a field: The pointer of



All edges have the same priority. The tree edges have the higher priority. Using Anchor Points for Structs.

Fig. 3. Priorities and Anchor Points

that field is just represented by the edge. In a practical example, the fields are visualized by text lines of labels, and the anchor points of edges are realized by the points at the nodes that are adjacent to these text lines (see fig. 3, right). As simplification, we allow to specify only the anchor points `unspecified` (these edges are anchored at the top or bottom of a node), `left`, `right` (anchored anywhere at the left/right side, see fig. 1) and `line` i which indicates that it is anchored at the left or right side of a node adjacent to the ith line of the node's label.

It is straightforward to applying the folding operations. The result is a flat graph $G = (V, E)$, where V contains only the visible nodes and E contains only the visible edges.

3 Partitioning of Nodes and Edges

3.1 A n-Level Hierarchy

After the application of the folding operations, the first phase of the layout algorithm partitions the visible nodes into levels. An integer rank $R(v)$ is calculated for each node $v \in V$. All nodes with the same rank form a level and are laid out on the same vertical position (see [8] for details).

Thus, it is appropriate to give nodes the same rank, if they are connected by edges anchored on the left or right side. These nodes can be drawn as neighbors such that the edge is a straight horizontal line between them (fig. 1). Of course, this is done if only one of such neighbor nodes at the left or right side exist. During the partitioning, these neighbor nodes are therefore fused into one node.

It is simpler to find the layout of the edges, if all edges are directed downwards and no edge crosses several levels. Thus, we build a proper hierarchy [13][14] by reverting upward edges and by splitting edges crossing several levels into small edges and dummy nodes. Reverted edges are marked such that arrowheads will later show the original direction.

Definition 2 *A* **proper** $n+1$**-level hierarchy** *is a directed graph* $G = (V, E)$

which satisfies the following conditions:

- *V is partitioned into $n+1$ disjoint subsets, i.e., $V = V_0 \cup V_1 \cup \ldots \cup V_n$, and $V_i = \{v \mid R(v) = i\}$.*
- *E is partitioned into n disjoint subsets, i.e., $E = E_0 \cup E_1 \cup \ldots \cup E_{n-1}$, and $E_i \subseteq V_i \times V_{i+1}$.*

3.2 Reduction of Crossings

The order of the nodes within a level determines the edge crossings in the layout, and a good ordering is one with few crossings. The problem of minimization of edge crossings is NP-complete [6], thus we use heuristics. The nodes are iteratively reordered within the levels according to the barycenter weights [13], or alternatively according to the median weights [5]. The heuristics stops if the number of crossings is not anymore reduced.

For the speed of the heuristics is most important, that the number of crossings can be calculated very fast. The original approach in [13] and [14] uses interconnection matrices, such that the calculation of crossings of edges between level i and $i+1$ needs time $O(|V_i|^2\ |V_{i+1}|^2)$. In the full paper, we present a representation of the levels such that the number of crossings can be calculated in time $O(|V_i| + |V_{i+1}| + |E_i| + c)$, where c is the number of crossings. A similar plane sweep algorithm is described in [3], however for the more general case of arbitrary line segments in the plane with runtime $O((|E| + c)\ log|E|)$.

4 Calculation of Coordinates

The y coordinates are canonical derived such that the center of the nodes of each level have the same vertical position. Even if after the crossing reduction, it is clear in which horizontal order the nodes must appear, it is still a difficult task to find x coordinates for them. In order to achieve a balanced layout, a node v must be placed in the middle of all adjacent nodes with edges to v. Good result come from the spring embedder or rubber band network algorithms, for instance [10] and [7]: The positions of nodes is determined by the forces imposed upon the nodes, e.g. because edges pull on the nodes similar to springs or rubber bands according to their priorities. While these algorithms try to place all nodes directly on the plane, which often results in a nonhierarchical layout and needs a long runtime, we use a similar method to improve the existing hierarchical layout. Because in our case, the ordering of the nodes is already fixed, the situation is much simpler, thus the speed is reasonable.

4.1 The Pendulum Method

The idea of the pendulum method: the nodes are the balls and the edges are the strings. If the uppermost balls are fixed on a ceiling, the balls on the strings swing to a balanced layout driven by their gravity, e.g., a ball of a level $i+1$ that is fixed by strings to two balls at level i will swing to a horizontal position just in the middle of these two balls, because the gravity imposes a horizontal force upon the ball proportional to the angle of the strings. Because the vertical position of the nodes is already fixed, the pendulum movement of the balls is simplified to horizontal movements, and the angle force is approximated by the horizontal deflection of the edges adjacent to a node (ball). The deflection of a ball is positive if it is pulled to the right by the deflection of the strings it hangs on, and negative if it is pulled to the left.

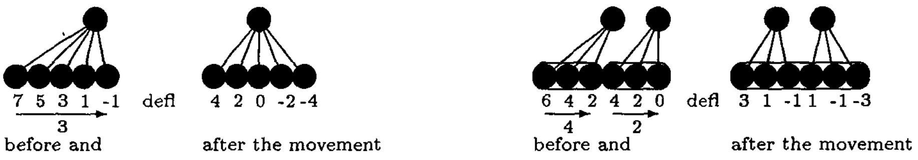


Fig. 4. Region movements and combinations

If several neighbored balls hang touching on the same ball, they influence each other: they cannot be placed all at the same point such that the deflection of each ball becomes zero, thus they move into a position such that the summary deflection of all these balls is zero, even if some balls still have a positive or negative deflection (fig. 4 left). We call the set of nodes influencing each other a *region*. We may have two touching regions whose left region is pulled to the right and whose right region is pulled to the left. In this case, these regions start to influence each other, i.e. they create a new region that consists of both. If we have the converse case, both regions are drawn asunder, they cannot be combined. Two touching regions that are pulled into the same direction can be combined, if the force of the one region is larger than the force of the other region such that the first region influences the second region (fig 4 right). If two regions are separated by horizontal space, they are independent and cannot be combined. However, they may move during the pendulum steps such that the separating space disappears, then they touch together, i.e. after the movement they may create a new, combined region. Thus, the pendulum method is an iterative process: we continuously traverse all levels, for each level, the regions are calculated and all nodes of a region are moved by the same horizontal offset according to the deflection value D_{pred} of the region. Starting from an initial assignment of a x coordinate $x(v)$ to each node v, this is done until the layout is balanced.

Definition 3 *The* **predecessor deflection** *of an edge $e = (s,t)$, a node v and a region $\{v_1, \ldots, v_m\}$ in a proper n-level hierarchy are defined as*

$$D_{pred}(e) = p(e)\ \ (x(s) - x(t))$$

$$D_{pred}(v) = \frac{\sum_{(w,v)\in E} D_{pred}((w,v))}{\sum_{(w,v)\in E}\ p((w,v))}$$

$$D_{pred}(\{v_1, \ldots, v_m\}) = \frac{\sum_{i\in\{1,\ldots,m\}} D_{pred}(v_i)}{m}$$

$p(e)$ ($p((w,v))$, respectively) denotes the priority of the edge $e = (w,v)$. Mathematically, the deflection of edges is a variant of the 1-norm.

After a top down traversal that balances the nodes, it is also possible to add a bottom up traversal using successor deflections D_{succ} that can be defined analogously. Physically, this corresponds to a rotation of the pendulum by 180 degree and fixing the former lowermost balls on the ceiling. The VCG tool performs top down traversals and bottom up traversals in alternating order.

As in the reality, a pendulum may start to oscillate. Even if this is very seldom, it is important to have a good decision function that stops the oscillation. Here, we use a weight that represents the sum of all forces on all nodes:

$$\sum_{v \in V} \frac{\sum_{(v,w)\in E} p((v,w))\ (x(w)-x(v)) + \sum_{(w,v)\in E} p((w,v))\ (x(w)-x(v))}{\sum_{(v,w)\in E} p((v,w)) + \sum_{(w,v)\in E} p((w,v))}$$

This value decreases with a high probability, because each step of a traversal reduces one summand very much while other summands may increase only a little bit. If the pendulum starts to oscillate, it does not decrease anymore: in this case the amount of increasing summands is equal or larger than the amount of decreasing summands. Then, the algorithm stops.

4.2 The Rubber Band Method

The pendulum method creates a balanced layout where all nodes have enough space to move to the left or to the right. However, the pendulum method does not force straight edges, if they are split into sequences of polygon segments and dummy nodes, because it analyzes only the predecessor edges of a node, or the successor edges of a node separately. When there is enough space between the nodes, it is appropriate to apply the rubber band method: as rubber bands, the predecessor and successor edges pull on the node at the same time such that the node is centered in order to eliminate the forces of different directions. Each node is moved by $W(\mathrm{v})$. Thus, dummy nodes, which have exactly one predecessor and one successor edge of the same priority, are forced to be positioned such that the gradient of both edges becomes equal: the combination of both edges appears as a straight line. In the rubber band method, neighbored nodes do not influence each other. A node is only moved to a new x coordinate if there is space enough around the node. Hence, there is no tendency to oscillate.

Definition 4 *The* **force weight** *of a node v is defined by*

$$W(v) = \frac{\sum_{(v,w)\in E} p((v,w))\ (x(w)-x(v)) + \sum_{(w,v)\in E} p((w,v))\ (x(w)-x(v))}{\sum_{(v,w)\in E} p((v,w)) + \sum_{(w,v)\in E} p((w,v))}$$

The rubber band method is a fine tuning phase that optimizes the horizontal positions of nodes. Since the reduction of $W(v)$ for each node v implies the reduction of the sum of all defections, we can use the same stop criterion as with the pendulum method.

4.3 Remarks About the Speed

The order of the nodes that are selected for repositioning influences the speed. A movement of a node may prevent other nodes to be moved, because nodes should not overlap: their surrounding space may disappear. As general hint, nodes should be preferred that must move very much. If more nodes tend to move to the right than to the left, the movements to the right should be done first.

In the pendulum method, neighbored node may influence each other, while the rubber band method ignores these influences. A combined method is also possible: The weights $W(v)$ are used to create regions as in the pendulum method. The result is a balanced layout that also forces straight edges. However, in our experience, this combined method is much slower than the sequence of the two different methods. The reason is that the weights $W(v)$ tend to be globally balanced while there are only small regions in the layout where $W(v)$ is not balanced. This imbalance must often be propagated over all nodes to get a complete balanced layout, i.e. in order to balance the local region, the whole layout must be changed. The pendulum method, as described, quickly forces parent nodes to be centered to children nodes, because an imbalance of the weights D_{pred} and D_{succ} that are defined only in the predecessor or the successor direction is more probable. Thus, the initial layout changes much more if the weights D_{pred} and D_{succ} are used, and hence, the propagation of imbalances of regions succeeds faster and results faster in a complete balancement.

5 Layout of Edges

Most graph layout tools do a good job when calculating the positions of nodes, but oversimplify the positioning of edges: Edge segments are drawn as straight lines from the border of the source node to the border of the target node. In fact, this results in readable pictures of the graph, if all nodes have the same shape and size, because edges are already split into edge segments and dummy nodes to form a proper hierarchy, thus it is not probably that an edge overlaps a node, because the dummy nodes and normal nodes do not overlap.

In our applications of the VCG tool, nodes very often have different sizes. Solving the layout constraint *'no edge goes through a visible node'* is here much more complex. Edge segments must be bent to get around the large nodes. This sequence of bendings can be drawn by straight polygon segments, or optionally as splines. In [8], a good, but complex spline routine is described that solves this problem. For efficiency reasons, our spline drawing routine is much simpler. The details of the layout of edges are described in the full paper.

6 Layout of Anchored Nodes

The layout of a proper hierarchy is appropriate, if edges ar anchored at the top or bottom of the nodes, because the edges are split into polygon segments pointing downwards. Upward edges are marked by drawing the arrowhead at the

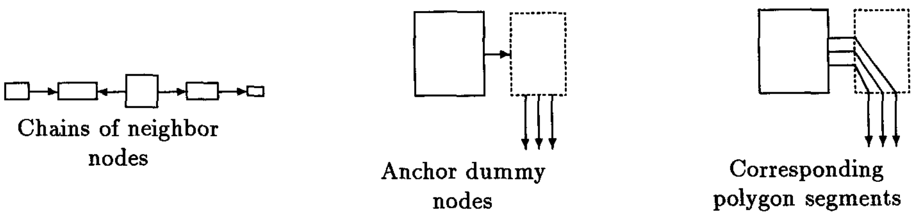


Fig. 5. Edges anchored `line` i

uppermost beginning of the polygons. Edges with anchors **`left`**, **`right`** and **`line`** i must be converted before crossing reduction. The VCG tool restricts each node to have at most one edge anchored left and one edge anchored right to the node. Nodes connected by such edges form a linear chain (fig. 5, left). These chains are fused into one node before the crossing reduction, because crossing reduction works only if the hierarchy is proper. Before the positioning of nodes, the chains are expanded again: As the result, the nodes of the chains are really neighbored at the same level and the left/right edges can be drawn as short straight edges.

Now, Edges anchored **`line`** i can be converted into an edge anchored **`right`**: We add an 'anchor dummy node' v for each node s with such edges and replace the edges (s, t) anchored **`line`** i by edges (v, t) (anchor point **`unspecified`**) and one edge (s, v) anchored **`right`** (fig. 5, middle). This situation now can be treated as above. After the positioning of the nodes, the anchor dummy node v is replaced by edge segments that connect s with the starting points of the edges on v, i.e. when drawing, the anchor dummy nodes are invisible, and their image is a sequence of edge segments (fig. 5, right).

7 Appearance of Objects

Further possibilities to influence the appearance of objects are shown in figure 6 which visualizes the dependences of shell programs (from [8]). Edges may be solid, dotted or dashed and may have different colors and sizes. The shape of a node may be a box, a rhomb, a triangle or an ellipse. It is possible to specify their rank and order within the levels. This allows to place the shells at the same rank as their birth dates, and to place the time axis at the left side of the graph. To avoid that the components of the graph are layouted separately, the parts of the graph are connected by some invisible edges. Invisible edges, as all other edges, influence the positions of the nodes as they would pull their adjacent nodes together. To avoid this effect for the invisible edges, we set the priority of the invisible edges to zero and the priority of the visible edges to 100. Finally, edges are drawn as splines.

The label of a node may be too small to contain all the information needed for the node. In this case, it is possible to specify text fields that are only shown on demand. A variant of this method is very useful, if the graph is scaled so much that the text label is unreadable. By selecting a node, its label and its additional text fields are shown in a readable size.

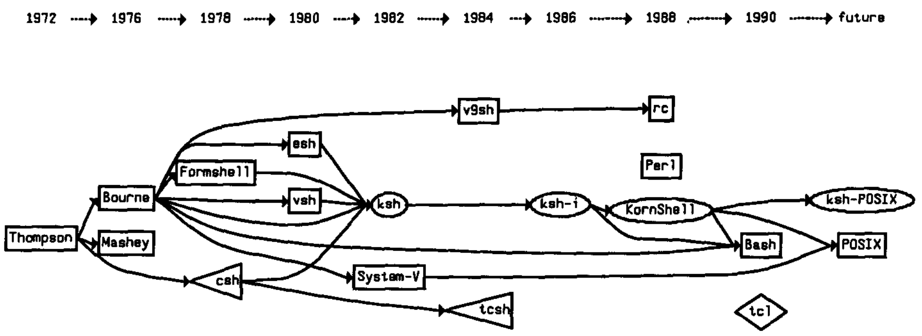


Fig. 6. Different Forms

The VCG tool is designed as an auxiliary tool to support debugging of data structures. In such an environment, the program to be inspected may produce a sequence of graphs that must be visualized. Thus, an animation interface is integrated into the VCG tool. The VCG tool and the program run concurrently and communicate by signals and time stamps of the file to detect when the new instance of the graph is produced.

8 Experiences and Statistics

Table 1 shows the performance of the different phases of the VCG tool. All measurements are done on a Sun Sparc 10/30 (32 MB mem., X11R5). Graph 1 is the visualization of a LR deterministic automaton produced by the TrafoLa parser generator [9]. Graph 2 is an intermediate representation of a program in the COMPARE compilation system. Graph 3 is an intermediate representation of a larger program. Graph 4 and 5 are complete graphs, i.e. all nodes are connected pairwise. Tree 1 is a syntax trees with attribute annotations, tree 2 is a binary tree of 12 levels, and tree 3 is a ternary tree of 8 levels.

It is not clear, whether the barycenter weight or the median weight results in better layouts. The time for the calculation of the median weight depends much more on the degree of nodes (see graph 4 and 5), because all adjacent nodes of v must be sorted to calculate the median value of a node v. Thus, barycentering has advantages if the average degree of nodes is large. Typically, barycentering results in a more symmetrically ordering of nodes within the levels. Thus, the pendulum method needs sometimes much more time to create a balanced layout after using the median weight (see t_{xy} in the graph examples).

As we see in graph 4, 5 and tree 2 and 3, the maximal size m of a level and the number of edges segments e influence t_c and t_{xy} very much. The reduction of crossings and the optimal placement of the nodes are the bottleneck during the layout. Thus, the VCG tool has options to control the maximal number of iterations during these phases: It is possible to create an unbalanced layout with very much crossings in very fast time. Using this feature, very complex graphs

Example	*Nodes*	*Edges*	*Alg.*	t_p	t_r	t_c	t_{xy}	t_d	t_{hand}	n	e	m	*Levels*	*Crossings*
Graph 1	132	288	Bary	0.23	0.05	3.28	0.99	0.03	5.5	648	796	43	33	530
			Median	0.23	0.05	1.48	11.3	0.05	16.0	648	796	43	33	576
Graph 2	66	95	Bary	0.10	0.01	0.34	0.33	0.42	3.0	245	274	25	20	53
			Median	0.11	0.01	0.56	0.64	0.17	3.5	245	274	25	20	86
Graph 3	417	808	Bary	0.77	0.41	142.5	40.3	0.32	190.0	6418	6809	233	72	12476
			Median	0.74	0.42	87.4	56.7	0.42	151.0	6418	6809	233	72	12896
Graph 4	23	253	Bary	0.12	0.11	17.6	3.64	0.35	22.5	1794	2024	122	23	3814
			Median	0.10	0.11	19.9	4.14	0.43	25.5	1794	2024	122	23	3652
Graph 5	26	325	Bary	0.15	0.17	15.4	5.16	0.32	22.5	2626	2925	157	26	6355
			Median	0.15	0.16	37.7	7.12	0.12	45.0	2626	2925	157	26	5981
Tree 1	2763	2762	Bary	2.64	0.73	1.10	5.78	0.15	11.5	2763	2762	56	132	0
Tree 2	4095	4094	Bary	6.38	1.97	0.46	3.67	0.24	13.5	4095	4094	2048	12	0
Tree 3	3280	3279	Bary	4.20	2.19	0.34	3.81	0.27	12.0	3280	3279	2187	8	0

Times (sum of user time and system time, measured by the computer): t_p = parsing (and folding), t_r = creation of a proper hierarchy, t_c = reduction of crossings, t_{xy} = assignment of coordinates, t_d = drawing. The complete runtime is the sum $T = t_p + t_r + t_c + t_{xy} + t_d$. The real time t_{hand} we had to wait until the graph was laid out and drawn is measured by hand. All times are measured in seconds.
n is the number of nodes laid out, including real nodes and dummy nodes. e is the number of segments to represent the edges. m is the max. number of nodes per level.

Table 1. Statistics

can be visualized in few seconds (e.g. graph 3 in 20 seconds). In such situations, the aesthetic quality is of minor interest as long as the graph can be inspected by following edges and centering nodes.

9 Conclusion

VCG is a tool that allows to visualize complex graphs in a compact way and in good performance. It can deal with many different kind of graphs including pointer networks of structs. Thus, it is very well appropriate to help on debugging data structures. It allows to fold parts of the graph, and to influence the layout to a large degree. We have described the layout algorithms, which are rather simple and use heuristics, but they are very fast and enable to explore very large graphs in reasonable time. Further work might address the following:

- Improve the stability of the layout: Similar graphs must create similar pictures.
- Allow incrementality of the layout: Adding an edge or a node to an layout should not cause a complete relayout.
- Allow different layout algorithms for nested subgraphs: Each subgraph can be laid out by a specialized variant of a layout algorithm. The combination of such algorithms must be analyzed.
- Improve the spline drawing routine.

The implementation of the tool is based on the diploma thesis of Iris Lemke [11] (VCG for SunView). It runs with SunView and X11 on many different platforms (SunOS, IRIX, IBM AIX, HP-UX. ...) and can produce different forms of output (PostScript, PBM, PPM). The tool is available via anonymous ftp at `ftp.cs.uni-sb.de` (134.96.7.254) in the directory /pub/graphics/vcg.

10 Acknowledgement

I wish to thank the COMPARE Consortium for the permission to distribute the VCG tool freely, and the anonymous referees for their helpful comments.

References

[1] Alt, M.; Aßmann, U; Someren, H: Cosy Compiler Phase Embedding with the CoSy Compiler Model, *in* Fritzson, P.A.: Compiler Construction, 5th International Conference, CC '94, Proceedings, Lecture Notes in Computer Science 786, pp. 278-293, Springer Verlag 1994

[2] Battista, G.D.; Eades, P.; Tamassia, R.: Algorithms for Drawing Graphs: An Annotated Bibliography, *avail. ftp at* `wilma.cs.brown.edu`, `/pub/gdbiblio.tex` 1993; *a prev. version was avail. as* technical report CS-89-09, Brown University, Department of Computer Science, Providence RI, Oct. 1989

[3] Bentley, J.L.; Ottmann, T.A.: Algorithms for Reporting and Counting Geometric Intersections, IEEE Trans. on Computers, Vol. C 28, No. 9, pp. 643-647, 1979

[4] Brandenburg, F.J.: Nice Drawings of Graphs are Computationally Hard, Visualization in Human Computer Interaction, Lecture Notes in Computer Science 439, pp. 1-15, Springer Verlag 1990

[5] Eades, P.; Wormald N.: The median heuristic for drawing 2-layers networks, technical report 69, Department of Computer Science, University of Queensland, 1986

[6] Eades, P.; McKay B.; Wormald N.: On an edge crossing problem, Proc. 9th Australian Computer Science Conf., pp. 327-334, 1986

[7] Fruchterman, T.M.J.; Reingold, E.M.: Graph drawing by forcedirected placement, Software – Practice and Experience, Vol. 21, pp. 1129-1164, 1991

[8] Gansner, E.R.; Koutsofios, E.; North, S.C.; Vo, K.: A Technique for Drawing Directed Graphs, IEEE Trans. on Software Engineering, Vol. 19, No. 3, pp. 214-230, March, 1993

[9] Heckmann, R.; Sander, G.: TrafoLa-H Reference Manual, *in* Hoffmann, Berthold; Krieg-Brückner, Bernd, Editors: Program Development by Specification and Transformation, Lecture Notes in Computer Science 680, Springer Verlag 1993

[10] Kamada, T.; Kawai, S.: An algorithm for drawing general undirected graphs, Information Processing Letters 31, pp. 7-15, 1989

[11] Lemke, I.: Entwicklung und Implementierung eines Visualisierungswerkzeuges für Anwendungen im Übersetzerbau, Universität des Saarlandes, Saarbrücken, Germany, Fachbereich 14 Informatik, *(to appear, in German)* 1994

[12] Paulisch, F.N.; Tichy, W.F.: EDGE: An Extendible Graph Editor, Software – Practice and Experience, Vol. 20, No. S1, pp. 63-88, June 1990

[13] Sugiyama, K.; Tagawa, S.; Toda, M.: Methods for visual understanding of hierarchical system structures, IEEE Trans. on Systems, Man, and Cybernetics SMC-11, No. 2, pp. 109-125, Feb. 1981

[14] Warfield, N.J.: Crossing theory and hierarchy mapping, IEEE Trans. on Systems, Man, and Cybernetics SMC-7, No. 7, pp. 505-523, Feb. 1977

[15] Wilhelm, Reinhard; Maurer, Dieter: Übersetzerbau: Theory, Konstruktion, Generierung, Springer Verlag 1992, *English Version to appear with Addison Wesley*