[Hen92] Henry, Tyson Rombauer: Interactive Graph Layout: The Exploration of Large Graphs, Ph.D. Dissertation, *available as* technical report TR-92-03, The University of Arizona, Department of Computer Science, June, 1992

[HeSa93] Heckmann, Reinhold; Sander, Georg: 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

[Him89] Himsolt, Michael: GraphEd: An Interactive Graph Editor, *in* Proc. STACS 89, Lecture Notes in Computer Science 349, pp. 532-533, Springer Verlag 1989

[KaKa89] Kamada, T.; Kawai, S.: An algorithm for drawing general undirected graphs, *in* Information Processing Letters 31, pp. 7-15, 1989

[KoEl91] Koutsofios, Eleftherios; North, Stephen C.: Drawing graphs with dot, technical report, AT&T Bell Laboratories, Murray Hill NJ, 1992

[Lem94] Lemke, Iris: Entwicklung und Implementierung eines Visualisierungswerkzeuges für Anwendungen im Übersetzerbau, Universität des Saarlandes, Saarbrücken, Germany, Fachbereich 14 Informatik, *(to appear, in German)* 1994

[Meh84a] Mehlhorn, Kurt: Data Structures and Algorithms 1: Sorting and Searching, Springer Verlag 1984

[Meh84b] Mehlhorn, Kurt: Data Structures and Algorithms 2: Graph Algorithms, and NP-Completeness, Springer Verlag 1984

[Meh84c] Mehlhorn, Kurt: Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry, Springer Verlag 1984

[PaTi90] Paulisch, Frances Newbery; Tichy, W.F.: EDGE: An Extendible Graph Editor, *in* Software – Practice and Experience, Vol. 20, No. S1, pp. 63-88, June 1990

[PrSh85] Preparata, Franco P.; Shamos, Michael Ian: Computational Geometry: An Introduction, Springer Verlag 1985

[STM81] Sugiyama, Kozo; Tagawa, Shojiro; Toda, Mitsuhiko: Methods for visual understanding of hierarchical system structures, *in* IEEE Transactions on Systems, Man, and Cybernetics SMC-11, No. 2, pp. 109-125, Feb. 1981

[War77] Warfield, N.J.: Crossing theory and hierarchy mapping, *in* IEEE Transactions on Systems, Man, and Cybernetics SMC-7, No. 7, pp. 505-523, Feb. 1977

[WiMa92] Wilhelm, Reinhard; Maurer, Dieter: Übersetzerbau: Theory, Konstruktion, Generierung, Springer Verlag 1992, *English Version to appear with Addison Wesley*

ple 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 [Lem94] (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.

References

[AAS94] 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

[BET93] Battista, Guiseppe Di; Eades, Peter; Tamassia, Roberto: Algorithms for Drawing Graphs: An Annotated Bibliography, *available via ftp from* `wilma.cs.brown.edu`, *file* `/pub/gdbiblio.tex` 1993 *a previous version was* Eades, Peter; Tamassia, Roberto: Algorithms for Drawing Graphs: An Annotated Bibliography, technical report CS-89-09, Brown University, Department of Computer Science, Providence RI, Oct. 1989

[BeOt79] Bentley, Jon L.; Ottmann, Thomas A.: Algorithms for Reporting and Counting Geometric Intersections, *in* IEEE Transactions on Computers, Vol. C 28, No. 9, pp. 643-647, 1979

[Bra90] Brandenburg, F.J.: Nice Drawings of Graphs are Computationally Hard, *in* Visualization in Human Computer Interaction, Lecture Notes in Computer Science 439, pp. 1-15, Springer Verlag 1990

[ChTa76] Cheriton, D.; Tarjan, R.E.: Finding Minimum Spanning Trees, *in* SIAM Journal of Computing 5, pp. 724-741, 1976

[EaWo86] Eades, P.; Wormald N.: The median heuristic for drawing 2-layers networks, technical report 69, Department of Computer Science, University of Queensland, 1986

[EMKW86] Eades, P.; McKay B.; Wormald N.: On an edge crossing problem, *in* Proc. 9th Australian Computer Science Conf., pp. 327-334, 1986

[FrWe93] Fröhlich, Michael; Werner, Mattias: Das interaktive Graph Visualisierungssytem daVinci V1.2, technical report *(in German),* University of Bremen, Germany, Fachbereich Mathematik und Informatik

[FrRe91] Fruchterman, T.M.J.; Reingold, E.M.: Graph drawing by forcedirected placement, *in* Software – Practice and Experience, Vol. 21, pp. 1129-1164, 1991

[GaJo83] Garey, M.R.; Johnson, D.S.: Crossing Number is NP-complete, *in* SIAM Journal of Algebraic and Discrete Methods, Vol. 4, No. 3, pp. 312-316, 1983

[GKNV93] Gansner, Emden R.; Koutsofios, Eleftherios; North, Stephen C.; Vo, Kiem-Phong: A Technique for Drawing Directed Graphs, *in* IEEE Transactions on Software Engineering, Vol. 19, No. 3, pp. 214-230, March, 1993

[GNV88] Gansner, Emden R.; North, Stephen C.; Vo, Kiem-Phong: DAG – A program that draws directed graphs, *in* Software – Practice and Experience, Vol. 17, No. 1, pp. 1047-1062, 1988

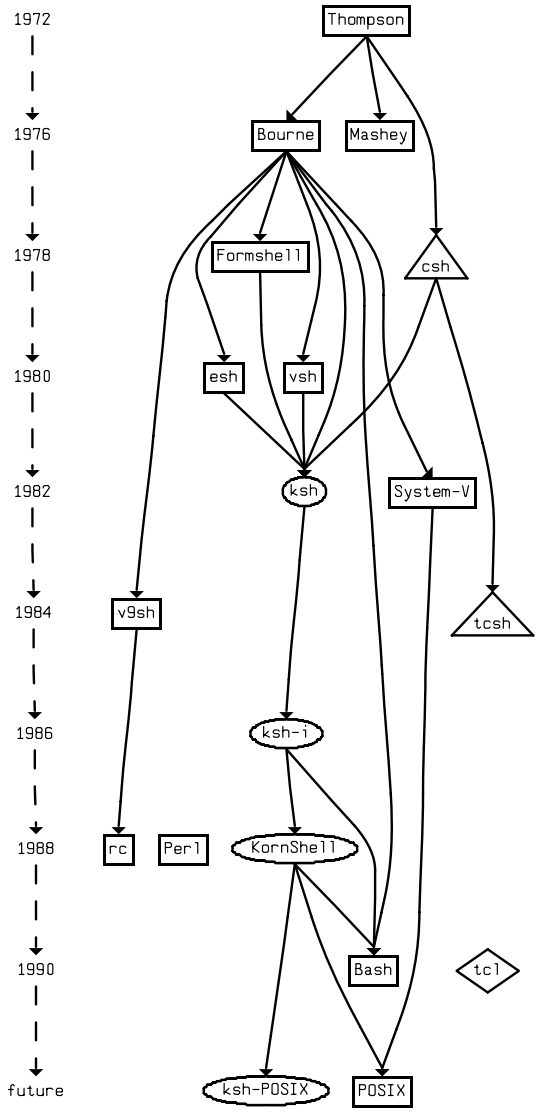


Figure 23: Different Forms

The times for parsing and rank assignment are obviously independent of the selected layout algorithms. The time for the additional fine tuning phase influences t_r not very much. For trees, the fine tuning phase has no effect. However, the omission of the fine tuning phase for real graphs leaves more dummy nodes and more edge segments, such that the layout is less readable.

It is not clear, whether the barycenter weight or the median weight results in better layouts. For the trees, the quality and speed of both methods is equal: they remove all crossings and result in nearly the same layout. For the graphs, the median weight removes more crossings in 5 cases, while the barycenter weight is better in 6 cases. In 5 cases, crossing reduction by barycentering is considerably faster (t_c in g.3 Bft, Bnt, g.4 Bnt, g.6 Bft, g.7 Bft), in 4 other cases, the calculation of median weights is faster (g.2 Mft, Mnt, g.4 Mft, g.5 Mft). As we see for graph 5, graph 6 and graph 7, the time for the calculation of the median weight increases much more depending on the degree of nodes, because all adjacent nodes of v must be sorted in order to calculate the median value of a node v. Thus barycentering has advantages if the average degree of nodes is large.

Depending on the selected weights for crossing reduction, the times for the further processing of the graphs may differ. 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 g.1 Bnt/Mnt, g.2 Bft/Mft, g.4, g.5).

The times t_{dp} and t_{ds} to draw the graph depend on the part of the graph that is initially visible in the window, because for efficiency, only this part is drawn. The speed of polygon drawings is reasonable, while splines need a lot of times due to very suboptimal routines for drawing spline curves. As consequence, splines are only usable for very small graphs. Further, all edges of trees are straight, such that no splines need to be drawn, thus $t_{dp} \approx t_{ds}$ holds for these cases.

As we see in graph 4, 6 and 7 and tree 4, 5 and 6, the maximal size of a level and the number of edges 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, such that it is possible to create an unbalanced layout with very much crossings in very fast time. Using this feature, large and complex graphs can be visualized in few seconds (e.g. graph 4 in 20 seconds), and in such situations, the aesthetic quality is of minor interest as long as the graph can be inspected by following edges and centering nodes.

10 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 sim-

Example	*Alg.*	t_p	t_r	t_c	t_{xy}	t_e	t_{dp}	t_{ds}	t_{hand}	n	e	l	m	c
Graph 1	Bft	0.07	0.01	0.14	0.15	0.01	0.04	0.50	1.0	97	124	13	15	14
	Mft	0.07	0.01	0.15	0.10	0.01	0.07	0.44	1.5	97	124	13	15	18
	Bnt	0.08	0.01	0.12	0.12	0.01	0.01	0.50	1.5	103	130	13	16	14
	Mnt	0.08	0.01	0.11	1.69	0.01	0.11	0.67	3.0	103	130	13	16	15
Graph 2	Bft	0.23	0.05	3.28	0.99	0.09	0.03	11.9	5.5	648	796	33	43	530
	Mft	0.23	0.05	1.48	11.3	0.13	0.05	12.9	16.0	648	796	33	43	576
	Bnt	0.23	0.05	3.32	1.41	0.13	0.06	21.0	6.0	900	1043	33	46	576
	Mnt	0.24	0.05	2.79	1.98	0.12	0.04	20.7	6.0	900	1043	33	46	527
Graph 3	Bft	0.10	0.01	0.25	1.41	0.01	0.44	2.90	3.5	219	248	20	22	93
	Mft	0.11	0.01	0.32	0.27	0.02	0.36	3.03	2.5	219	248	20	22	78
	Bnt	0.10	0.01	0.34	0.33	0.02	0.42	3.57	3.0	245	274	20	25	53
	Mnt	0.11	0.01	0.56	0.64	0.03	0.17	3.32	3.5	245	274	20	25	86
Graph 4	Bft	0.77	0.41	142.5	40.3	9.30	0.32	915.3	190.0	6418	6809	72	233	12476
	Mft	0.74	0.42	87.4	56.7	8.00	0.42	964.7	151.0	6418	6809	72	233	12896
	Bnt	0.75	0.44	118.1	37.4	11.7	0.39	1144.5	168.0	7225	7616	72	304	16563
	Mnt	0.76	0.46	146.7	55.8	14.2	0.44	1201.2	216.0	7225	7616	72	304	16916
Graph 5	Bft	0.12	0.08	9.75	2.32	0.09	0.44	29.7	14.0	1160	1330	20	91	1980
	Mft	0.09	0.08	4.41	20.9	0.09	0.22	28.8	27.5	1160	1330	20	91	1882
Graph 6	Bft	0.12	0.11	17.6	3.64	0.17	0.35	60.9	22.5	1794	2024	23	122	3814
	Mft	0.10	0.11	19.9	4.14	0.15	0.43	60.6	25.5	1794	2024	23	122	3652
Graph 7	Bft	0.15	0.17	15.4	5.16	0.26	0.32	120.7	22.5	2626	2925	26	157	6355
	Mft	0.15	0.16	37.7	7.12	0.24	0.12	134.7	45.0	2626	2925	26	157	5981
Tree 1	Bft	0.63	0.04	0.27	3.57	0.02	0.56	0.56	7.0	614	613	36	38	0
	Mft	0.64	0.04	0.27	3.60	0.03	0.54	0.55	7.0	614	613	36	38	0
Tree 2	Bft	2.64	0.73	1.10	5.78	0.17	0.15	0.16	11.5	2763	2762	132	56	0
	Mft	2.68	0.72	1.11	5.75	0.17	0.15	0.17	11.5	2763	2762	132	56	0
Tree 3	Bft	2.01	0.49	0.22	1.80	0.05	0.12	0.14	5.5	2047	2046	11	1024	0
	Mft	1.99	0.50	0.23	1.82	0.07	0.13	0.15	5.5	2047	2046	11	1024	0
Tree 4	Bft	6.38	1.97	0.46	3.67	0.17	0.24	0.25	13.5	4095	4094	12	2048	0
	Mft	6.33	1.98	0.46	3.67	0.15	0.23	0.25	13.5	4095	4094	12	2048	0
Tree 5	Bft	4.20	2.19	0.34	3.81	0.12	0.27	0.29	12.0	3280	3279	8	2187	0
	Mft	4.21	2.19	0.37	3.83	0.13	0.26	0.29	12.0	3280	3279	8	2187	0
Tree 6	Bft	1.01	0.19	2.24	21.6	0.05	0.23	0.27	26.5	1115	1124	5	656	0
	Mft	1.02	0.19	2.23	21.6	0.03	0.23	0.26	26.5	1115	1124	5	656	0

Table 1: Statistics

The standard method for rank assignment is depth first search. The layout algorithm Bft and Mft have the additional fine tuning phase, while Bnt and Mnt ommit this phase. Bft and Bnt use the barycenter weight for crossing reduction, while Mft and Mnt use the median weight.

t_p is the time for parsing (and folding). t_r is the time for the creation of a proper hierarchy. t_c is the time for the reduction of crossings. t_{xy} is the time for the assignment of coordinates. t_e is the time for the calculation of edge bendings. t_{dp} is the time for the drawing. Edges are represented as polygon segments. t_{ds} is the time for the additional calculation of all splines and their drawing. All these times are the sum of user time and system time, measured by the computer.

The complete runtime is either the sum $T_P = t_p + t_r + t_c + t_{xy} + t_e + t_{dp}$, if polygon segments are used, or $T_S = t_p + t_r + t_c + t_{xy} + t_e + t_{ds}$, if splines are used. To compare with, t_{hand} is the real time T_P we had to wait until the graph was laid out and drawn. It 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. l is the number of levels, and m is the maximal number of nodes within a level. Finally, c is the number of crossings in the layout.

- Graph 3 is an intermediate representation of a program in the COMPARE compilation system (66 nodes, 95 edges).
- Graph 4 is an intermediate representation of a larger program (417 nodes, 808 edges).
- Graph 5, graph 6 and graph 7 are complete graphs, i.e. all nodes are connected pairwise. Graph 5 consists of 20 nodes and 190 edges, graph 6 of 23 nodes and 253 edges, and graph 7 of 26 nodes and 325 edges.
- Tree 1 and tree 2 are syntax trees with attribute annotations. Tree 1 consists of 614 nodes and 613 edges, and tree 2 of 2763 nodes and 2762 edges.
- Tree 3 is a binary tree of 11 levels: 2047 nodes, 2046 edges.
- Tree 4 is a binary tree of 12 levels: 4095 nodes, 4094 edges.
- Tree 5 is a ternary tree of 8 levels: 3280 nodes, 3279 edges.
- Tree 6 is a variant of a 4-ary tree with many neighbor nodes: 1115 nodes and 1114 edges.

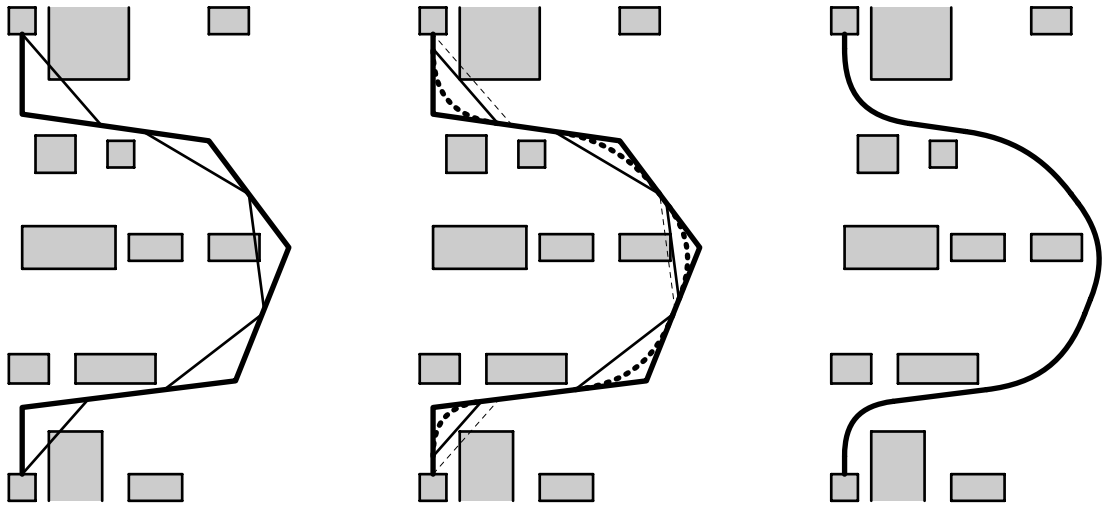


Figure 20: Spline calculation

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.

8 Appearance of Objects

Further possibilities to influence the appearance of objects are shown in figure 23 which visualizes the dependences of shell programs (from [GKNV93]). 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.

The VCG tool is designed as an auxiliary tool to support debugging of data structures. In a compilation 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. The program continuously produces graph specifications into a file while the VCG tool visualizes in parallel. In order to detect when the new instance of the graph is produced, the communication is done by signals and time stamps of the file.

9 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 with 32 MB memory and X11R5. (A SunView version exists, too.)
The examples are:

- Graph 1 is the visualization of a LR deterministic automaton produced by the TrafoLa parser generator (see [HeSa93]). It consists of 44 nodes and 75 edges.

- Graph 2 is a larger LR deterministic automaton (132 nodes, 288 edges).

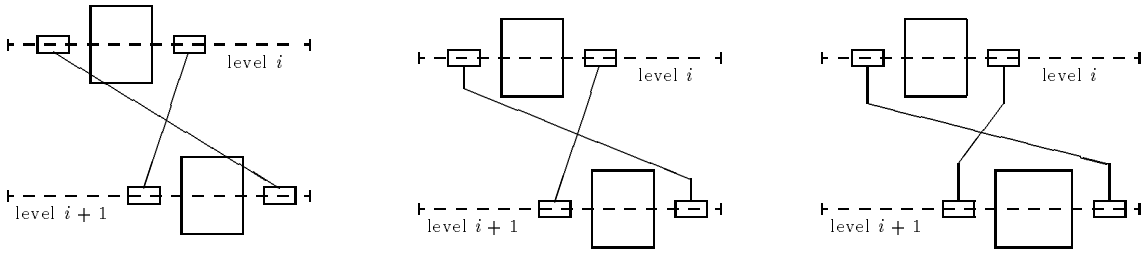


Figure 19: Situations without and with edge bendings

point. As simplification, we assume b=c, such that we have 3 control points that form a triangle that completely encloses the spline. We now compute the spline control points by looking for appropriate triangles at the bendpoints of the edge representing polygon. If these triangles do not overlap nodes, dummy nodes or other bendpoints, the spline lines do not cross nodes or other spline lines, except the crossings existed already with polygon segments. The middle control point is obviously the bendpoint itself, and the gradients at the start and end of the spline are determined by the adjacent polygon segments. Thus, we start with control points on the middle of the adjacent polygon segments such that an isoscele triangle is produced, and reduce the size of this triangle until all nodes and bendpoints are outside the triangle. Then, we draw the spline inside the triangle (figure 20). Because the spline is enclosed by the triangle whenever the control points b and c are on the sceles of the triangle, we can influence the sharpness of the curve bendings by correcting the control points.

Most important for the efficiency of this computation is the fact, that each bendpoint of a polygon segment belongs to exactly one level: Either the bendpoint is produced by a dummy node of that level, or it is a bendpoint of a vertical edge segment within this level. Thus, in order to check a triangle, only the nodes and bendpoints of the same level as the middle control point of the triangle must be inspected. Because the nodes within the levels are already sorted, normally a very small number of nodes which potentially overlaps the triangle must be checked.

7 Layout of Anchored Nodes

The layout of a proper hierarchy is appropriate, if edges are 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 uppermost beginning of the polygons. Edges with anchors `left`, `right` and `line` i must be converted before crossing reduction.

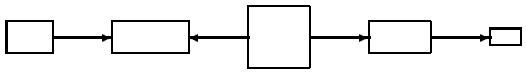


Figure 21: Chains of neighbor nodes

The VCG tool restricts each node to have at most one neighbor node at the left and at the right side, which implies that each node has at most one edge anchored left and one edge anchored right. Thus, the nodes connected by such edges form a linear chain (figure 21). These chains are fused into one node before the crossing reduction, because this 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.

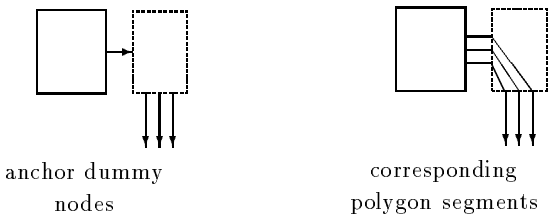


Figure 22: Edges anchored `line` i

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 $(s, t, \texttt{line}\ i, p, c)$ by edges $(v, t, \texttt{unspecified}, p, c)$ and one edge $(s, v, \texttt{right}, p, c)$ (figure 22). This situation now can

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 $W_4(v)$ are used to create regions, which must be fused according to the same criteria of the pendulum method, and all nodes of a region are moved by the same amount. 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 two different methods. The reason is that the weights $W_4(v)$ tend to be globally balanced while there are only small regions in the layout where $Force(v)$ 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.

6 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 problem (*'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 [GKNV93], a good, but complex spline routine is described that solves this problem. For efficiency reasons, our approach is much simpler.

6.1 Bendings of Edges

Even in a proper hierarchy, straight edges would be drawn through large neighbored nodes, if they start at very small nodes (figure 19, left). To avoid this, we allow each edge to be bent at two points (figure 19, middle). In consequence, an original edge, which is represented by a sequence of n small edges in the proper hierarchy, may be bent at each of its $n-1$ dummy nodes and additionally at $2n$ points within the edges. This worst case of $3n-1$ bendings occurs extremely seldom, because the previous layout phases force straight lines.

Since the nodes are distributed into layers and there is a minimum vertical space between the largest nodes of the layers, a correct layout would be to bend all edges such that they have vertical segments of the size of the largest node of the layer (figure 19, right). In this case, all nonvertical segments start at the same y coordinate and stop at the same y coordinate. They do not influence each other, i.e. they are easy to distinguish, but potentially to much bendings are produced. The graph looks much better if only the bendings are produced that are really necessary (figure 19, middle). However, a bending of an edge may cause a new crossing with a neighbored, straight edge. To avoid this, an edge is also bent if it is drawn through the vertical segment of a bent edge. Thus, the calculation of bendings is an iterative algorithm that looks for optimal bendings between the two extremes 'no edge is bent' and 'all edges are maximally bent'. To test the overlapping or additional crossings of an edge between level i and $i+1$, only the nodes at these levels and the edges between these levels must be inspected:

while $\exists e \in E$ *that overlaps a node*
or crosses a vertical segment of an edge e' **do**
enlarge the vertical segments of e*;*
od

6.2 Computing Splines

Finally, edges can be drawn as sequence of straight line segments between the bendpoints. This is appropriate for the interactive use of the tool where the picture must be refreshed very fast. If the picture of a graph must be printed on paper in high quality, it is worth to spend additional time to piecewise calculate Bezier splines to get smooth transitions between the edge segments.

Cubic Bezier splines have 4 control points a, b, c and d, such that a is the start point of the spline, d is the end point of the spline and the segments (a,b) and (c,d) give the gradient at the start and end

```
(1)  repeat
(2)      old_sum= D_sum;
(3)      traverse the levels top down and for level i do
(4)          let v_1, ..., v_n be the ordered sequence of the nodes of level i;
(5)          partition the nodes into a ordered sequence of regions R_1, ..., R_m with
(6)              R_j is a continuous subsequence {v_{i_j}, ..., v_{n_j}}
(7)              such that for all nodes v ∈ R_j the value D_pred(v) has the same sign
(8)              and all subsequent nodes v_k, v_{k+1} ∈ R_j are touching;
(9)          repeat
(10)             for subsequent, touching regions R_k, R_{k+1} do
(11)                 if D_pred(R_k) > 0 and D_pred(R_{k+1}) < 0 then
(12)                     replace R_k and R_{k+1} by the combined region R'_k = R_k ∪ R_{k+1};
(13)                 else if D_pred(R_{k+1}) ≥ 0 and D_pred(R_k) > D_pred(R_{k+1}) then
(14)                     replace R_k and R_{k+1} by the combined region R'_k = R_k ∪ R_{k+1};
(15)                 else if D_pred(R_k) < 0 and D_pred(R_{k+1}) < D_pred(R_k) then
(16)                     replace R_k and R_{k+1} by the combined region R'_k = R_k ∪ R_{k+1};
(17)                 fi
(18)             od
(19)         until the sequence of regions does not change anymore;
(20)         for each region R_k do
(21)             if D_pred(R_k) > 0 then
(22)                 move all nodes of R_k to the right by
(23)                     min{⌊D_pred(R_k)⌋, space between R_k and R_{k+1}} units;
(24)             else if D_pred(R_k) < 0 then
(25)                 move all nodes of R_k to the left by
(26)                     min{⌊−D_pred(R_k)⌋, space between R_{k−1} and R_k} units;
(27)             fi
(28)         od
(29)     od
(30)     new_sum= D_sum;
(31) until old_sum≤ new_sum;
```

Figure 18: Pendulum Method

only moved to a new x coordinate if there is space enough around the node. Hence, there is no tendency to oscillate.

Definition 7 *The* **force weight** *of a node v is defined by*

$$W_4(v) = \frac{W_3(v)}{\sum_{(v,w,a,p,c)\in E} p + \sum_{(w,v,a,p,c)\in E} p}$$

The rubber band method is a fine tuning phase that optimizes the horizontal positions of nodes similar to the fine tuning phase at the rank assignment for the vertical positions. Here, a new position of a node is feasible if the node does not overlap with other nodes, and the order of the nodes within the levels has not changed. Since the reduction of $W_4(v)$ for each node v implies the reduction of D_{sum}, we can use this value again as stop criterion.

```
while ∃v ∈ V with W_4(v) > threshold do
    old_sum= D_sum;
    T = x(v) + ⌊W_4(v)⌋;
    if feasible(T, v) then x_neu(v) = T; fi
    new_sum= D_sum;
    if old_sum≤new_sum then break; fi
od
```

5.3 Remarks about the Speed

Note that in both methods the updating of D_{sum} can be done incrementally during the calculation of new positions of nodes to improve the efficiency. Further, 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 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

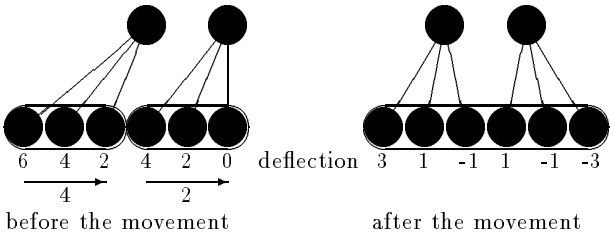


Figure 17: Region combinations

consists of both (figure 16). 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 (figure 17). 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 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 6 *Let (V, E) be a proper n-level hierarchy. The* **predecessor deflection** *of an edge $e = (s, t, a, p, c)$ is defined as*

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

Mathematically, the deflection of edges are variants of the 1-norm.
The predecessor deflection of a node v is defined as

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

Analogously, the predecessor deflection of a region $\{v_1, \ldots, v_n\}$ is

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

The pendulum method is sketched in figure 18 using the predecessor deflections. Nodes $v_1 = (w_1, h_1)$ and $v_2 = (w_2, h_2)$ are touching if they are separated by *min_dist*; they are detected by $x(v_1) + w_1 + a\ min\_dist \geq x(v_2)$. Regions R_1 and R_2 are touching, if the last node of R_1 touches the first node of R_2. 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. Top down traversals and bottom up traversals can be performed alternating.

Furthermore, 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:

$$D_{sum} = \sum_{v\in V} \frac{W_3(v)}{\sum_{(v,w,a,p,c)\in E}\ p + \sum_{(w,v,a,p,c)\in E}\ p}$$

where

$$W_3(v) \quad = \sum_{(v,w,a,p,c)\in E} p\ \ (x(w) - x(v)) + \sum_{(w,v,a,p,c)\in E} p\ \ (x(w) - x(v))$$

The value D_{sum} 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, D_{sum} 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.

5.2 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. 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

same level, whether the reversion of both nodes would remove crossings. This reduces the number of crossings again very much (see the statistics in [GNV88]).

5 Calculation of Coordinates

Even if after the crossing reduction, it is clear in which horizontal and vertical order the nodes must appear, it is still a difficult task to find (x, y) coordinates for them. The aesthetic criterion 'balancement' require that a node v is 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 [KaKa89] and [FrRe91]: 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.

If all edges pull on the nodes without constraints, the layout collapses to one point. To avoid this, a minimal distance *min_dist* between the nodes is necessary, and the positions of the outermost nodes of the rubber band must be fixed not too close together such that there is some freedom to place the nodes. Therefore, a prepass is necessary that creates a good initial layout from which the rubber band method starts. We use a pendulum method as prepass.

5.1 The Pendulum Method

First, we assign to each node v the y coordinate $y(v)$ such that the node is vertically centered at a virtual horizontal line that corresponds to a level (figure 14). The vertical distance of the levels is selected such that the highest nodes of two adjacent levels are still separated by some vertical space.

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.

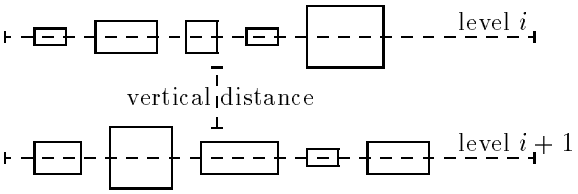


Figure 14: Vertical positioning at the levels

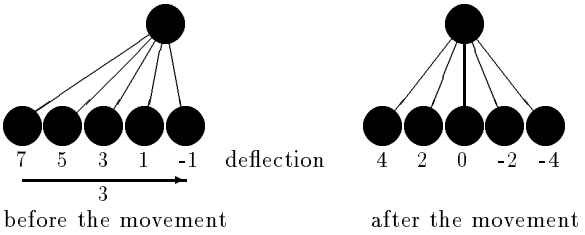


Figure 15: Region movements

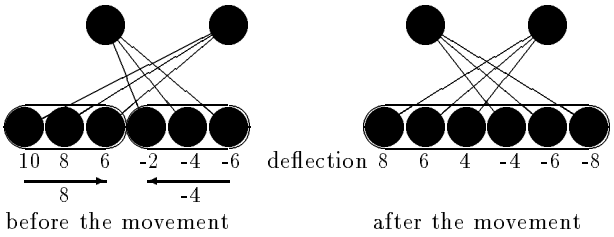


Figure 16: Region combinations

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 (figure 15). 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

```
(25) repeat
(26)     if the nodes v_i, ..., v_{i+k} of some level j of new_layout have the same weight_pred then
(27)         rotate(v_i, ..., v_{i+k});
(28)         traverse the levels starting from j top down and for each adjacent pair of levels do
(29)             resort the lower level of new_layout using weight_pred;
(30)         od
(31)         traverse the levels bottom up and for each adjacent pair of levels do
(32)             resort the lower level of new_layout using weight_succ;
(33)         od
(34)     fi
(35)     if crossings(new_layout)<old_nr_crossings then
(36)         old_layout=new_layout; old_nr_crossings=crossings(new_layout); have_alternative=no;
(37)     else if have_alternative then
(38)         new_layout=old_layout; have_alternative=no;
(39)         if the nodes v_i, ..., v_{i+k} of some level j of new_layout have the same weight_pred then
(40)             rotate(v_i, ..., v_{i+k});
(41)             traverse the levels starting from j top down and for each adjacent pair of levels do
(42)                 resort the lower level of new_layout using weight_pred;
(43)             od
(44)             traverse the levels bottom up and for each adjacent pair of levels do
(45)                 resort the lower level of new_layout using weight_succ;
(46)             od
(47)         fi
(48)         if crossings(new_layout)<old_nr_crossings then
(49)             old_layout=new_layout; old_nr_crossings=crossings(new_layout); have_alternative=no;
(50)         else if crossings(new_layout)>old_nr_crossings then
(51)             new_layout=old_layout; have_alternative=no;
(52)         fi
(53)     else if crossings(new_layout)=old_nr_crossings then have_alternative=yes;
(54)     else new_layout=old_layout; have_alternative=no;
(55)     fi
(56)     ... similar for the inverse traversal direction ...
(57) until old_layout has not changed anymore;
```

Figure 13: Reduction of Crossings, Part 2

crossings. This has further the advantage that if both layouts have equal crossings, we keep the old layout as a second alternative (backtrack point) for the case that further traversals on the new layout make it worse. This innovation improves the layout, because the iterations do not stop as long as a backtrack layout is available where we can start looking for better layout. At a third place, it may happen that nodes have exactly the same weight. Since we use a randomized quicksort [Meh84a] to sort the levels, the resulting positions of these nodes are arbitrary and not necessarily optimal. To improve this situation, we rotate these nodes and check after a rotation whether a traversal of the levels starting from this level results in a better layout. The resulting algorithm is sketched in figures 12 and 13.

The speed of the crossing reduction algorithm depends heavily on the implementation of the crossing calculation algorithm and on the implementation of quicksort. Since after some initial steps, the lists of nodes at the levels are presorted to a large degree, a randomized quicksort must be used. The lists of nodes are temporary stored into arrays, which are sorted and stored back into the lists. Further improvements are the detection that no crossings are anymore there in order to stop the algorithm, and the detection that a level has not changed when resorting it in order to stop the traversals.

Finally in this phase, the neighbor nodes that were fused before this phase are expanded again. At last, a local optimization phase is added that checks for each neighbored pair of nodes of the

```
(1)  have_alternative=no;
(2)  new_layout=old_layout;
(3)  old_nr_crossings=crossings(old_layout);
(4)  repeat
(5)      traverse the levels top down and for each adjacent pair of levels do
(6)          resort the lower level of new_layout using weight_pred;
(7)      od
(8)      if crossings(new_layout)<old_nr_crossings then
(9)          old_layout=new_layout; old_nr_crossings=crossings(new_layout); have_alternative=no;
(10)     else if have_alternative then
(11)         new_layout=old_layout; have_alternative=no;
(12)         traverse the levels top down and for each adjacent pair of levels  do
(13)             resort the lower level of new_layout using weight_pred;
(14)         od
(15)         if crossings(new_layout)<old_nr_crossings then
(16)             old_layout=new_layout; old_nr_crossings=crossings(new_layout); have_alternative=no;
(17)         else if crossings(new_layout)>old_nr_crossings then
(18)             new_layout=old_layout; have_alternative=no;
(19)         fi
(20)     else if crossings(new_layout)=old_nr_crossings then have_alternative=yes;
(21)     else new_layout=old_layout; have_alternative=no;
(22)     fi
(23)     ... similar for bottom up traversals ...
(24) until old_layout has not changed anymore;
```

Figure 12: Reduction of Crossings, Part 1

where $v_1, \ldots, v_n$ is the sequence of predecessors of v ordered according Pos. If n is odd, then $Median_{pred}(v)$ is $Pos(v_{\lceil \frac{n}{2} \rceil})$. In [STM81], the **barycenter weight** is proposed which is the arithmetic middle value of the positions of the predecessors:

$$Bary_{pred}(v) = \frac{\sum_{v_i \in predecessor(v)} Pos(v_i)}{|predecessor(v)|}$$

Both weights are appropriate to find orderings of the nodes with few crossings. While the median weight is theoretical well analyzed [EaWo86], barycentering happens to yield more symmetrical layouts. It is unclear which weight is more appropriate – depending on the graphs, the one or the other may be better – thus both weights are implemented in the VCG tool. Furthermore, if we assume the lower level to be fixed, we can also resort the upper level using similar weights:

$$Median_{succ}(v) = \frac{Pos(v_{\lfloor \frac{n}{2} \rfloor + 1}) + Pos(v_{\lceil \frac{n}{2} \rceil})}{2}$$

where $v_1, \ldots, v_n$ is the sequence of successors of v ordered according Pos and

$$Bary_{succ}(v) = \frac{\sum_{v_i \in successor(v)} Pos(v_i)}{|successor(v)|}$$

The algorithm now works as follows (where *weight* is *Median* or *Bary*):

```
traverse the levels top down and
  for each adjacent pair of levels do
    resort the lower level using weight_pred
od
traverse the levels bottom up and
  for each adjacent pair of levels do
    resort the upper level using weight_succ
od
```

Note that the result of these traversals depends mainly on the initial ordering at the first level, because the ordering of all other levels is inherited from it. There are several inappropriate situations: In the first place, the bottom up traversal resorts the first level at the end. In this case, a new top down traversal can improve the result further, such that both traversals must be iterated until a fixed point is reached. At the second place, a traversal does not necessarily result in fewer crossings because of the usage of heuristical weights. Thus, after each traversal, it must be checked, whether the new layout is better than the old layout. We store the old and the new layout in two buffers, and save the new layout into the old buffer only if it has fewer

```
(0)  procedure calc_crossings(level_i, level_{i+1})
(1)      nr_crossings= 0; size_of_UL= 0; size_of_LL= 0;
(2)      UL=empty; LL=empty;
(3)      for w in level_i ∪ level_{i+1} do last_occurrence(w) =nil; od
(4)      for w alternating from level_i and level_{i+1}, i.e., in the order of Ord do
(5)          if Ord(w) is odd then
(6)              k_1 = 0; k_2 = 0; k_3 = 0;
(7)              if last_occurrence(w) ≠ nil then
(8)                  for v in UL from the start of UL to last_occurrence(w) including in that order do
(9)                      if v = w then
(10)                         k_1 = k_1 + 1;
(11)                         k_3 = k_3 + k_2;
(12)                         delete v from UL;
(13)                         size_of_UL=size_of_UL−1;
(14)                     else k_2 = k_2 + 1;
(15)                     fi
(16)                 od
(17)                 nr_crossings=nr_crossings+k_1*size_of_LL+k_3;
(18)             fi
(19)             for all edges e with start point w in order according Ord of their end points do
(20)                 let w' = endpoint of e;
(21)                 Remark: Note that Ord(w) <Ord(w');
(22)                 Add w' at the end of LL;
(23)                 size_of_LL=size_of_LL+1;
(24)                 last_occurrence(w') =this new instance of w' in LL;
(25)             od
(26)         else Ord(w) is even
(27)             ...symmetrically ...
(28)         fi
(29)     od
(30) end
```

Figure 11: Calculation of Crossings

line (19). The challenge is to get the edges e starting at w just in the order of their endpoints, but without sorting the edges. This can be solved be sorting the edges in advance, i.e. before line (1): When the procedure starts, we traverse both levels according to decreasing *Ord*. We start with empty auxiliary adjacency list of the nodes. For each node w, we add the edges $(v, w, \ldots)$ with target w at the beginning of the auxiliary adjacency list of v. It follows that edges with target node w of higher $Ord(w)$ are later in the final adjacency list of v. The creation of the auxiliary adjacency lists needs $O(n_1 + n_2 + e)$ because the insertion at the beginning of an adjacency list can be performed in constant time. In line (14), we use these auxiliary adjacency lists to get the nodes in the right order. □

The number of crossings of the whole graph can now be calculated by the sum of the crossings of each adjacent pair of levels.

4.2 Reordering of Nodes

The reduction of crossings is done by resorting the doubly linked linear lists of nodes that represent the levels. The number of crossings between two levels depends on the order of the nodes within both levels. First, we assume, that the order of the upper level is fixed. Each node v has an attribute $Pos(v)$ indicating its position within the levels. Now we try to sort the lower level according to some weights. As basic idea of the heuristics, a node of the lower level must be placed anywhere in the middle of the nodes of the upper level with edges to it, because this implies that the edges go not too much cross. In [EaWo86] and [GKNV93], the (interpolated) **median weight** is proposed to find the middle point of nodes:

$$Median_{pred}(v) = \frac{Pos(v_{\lfloor \frac{n}{2} \rfloor + 1}) + Pos(v_{\lceil \frac{n}{2} \rceil})}{2}$$

level compared to the lower level. Thus we assume that the horizontal positions of the nodes are alternating: If we use a sweep line to traverse the levels in horizontal direction, the sweep line touches alternating a node from the upper level and one from the lower level. The Number $Ord(v)$ gives the horizontal position of v, i.e. $Ord(v)$ is odd for nodes v at the upper level, and even for nodes on the lower level. Edges are considered as directed from the left to the right, i.e. we call v the start node and w the end node of an edge between v and w, if $Ord(v) < Ord(w)$. (This artificial direction of edges is only to support the argumentation and has nothing to do with the direction of the edges in the final drawing.) In order to avoid to count crossings twice, we inspect the crossings of an edge when the sweep line touches its end point. If the end point w of an edge e belongs to the upper level, then e is crossed by all edges e' starting before w at the upper level and ending after w at the lower level (situation 1, figure 10). Note that crossings of e with edges e' that end before w are already counted. Furthermore, e is crossed by all edges starting before the start point v of e at the lower level, and ending after w (situation 2). For the end points of edges e at the lower level, the symmetrical case holds. No other crossings can occur.

Situation 1:
$Ord(v') < Ord(w) < Ord(w')$

Situation 2:
$Ord(v') < Ord(v) < Ord(w) < Ord(w')$

Figure 10: Crossing situations

We must store the active edges, i.e. the edges that start left to the sweep line, but end right to the sweep line: Therefore, we use two doubly linked lists UL and LL. UL contains the end nodes of edges of the upper level and LL contains the end nodes of edges of the lower level. Because a node may be end point of several edges, it may occur several times in these lists. The lists are sorted according to the order of the start nodes, i.e. if w is before w' in UL, then there is an edge with start node v and end node w and another edge with start node v' and end node w' and $Ord(v) < Ord(v')$ holds.
The calculation of crossings (see figure 11) is done in this way:

- Situation 1: If w in UL is touched by the sweep line, there are as many edges with endpoint w as occurrences of w are in UL. All these edges are crossed by all active edges with end points at the lower level, which can easyly counted by looking at the size of LL. Symmetrically for w in LL.
- Situation 2: For each occurrence of w in UL exist an edge e. All nodes $w' \neq w$ before this instance if w in UL are endpoints of edges that have started before the start point of e but end after w, because otherwise, they would have been removed from UL by becoming inactive. Symmetrically for w in LL.

To check how often w is in UL, it is appropriate to store the last occurrence of w in UL in a field *last_occurrence*. Then, we need not to traverse UL completely; it is sufficient to traverse it from the start to *last_occurrence*(w).,

Theorem 1 *The number of crossings c between e line segments starting at n_1 points at the upper level and ending at n_2 points at the lower level can be calculated in time $O(n_1 + n_2 + e + c)$.*

Proof: The correctness of the algorithm in figure 11 follows from the previous discussion. Note that in line (17) k_1 is the number of the occurrences of w in UL, hence k_1*size_of_LL is the number of crossings produced by situation 1. In line (11) k_2 is the number of predecessors of the instance of w in UL that is removed next. Thus, the number of crossings produced by the corresponding edge with endpoint w is k_2 according to situation 2. k_3 is the sum of all crossings produced by situation 2 for edges with endpoint w.
Because UL and LL are doubly linked lists, insertion at the end of the list and deletion of an element are performed in $O(1)$. The cost of line (1) - (3) is $O(n_1 + n_2)$. Line (4) - (6), (17), and (26), ...are executed $n_1 + n_2$ times. All nodes together are inserted into UL (or LL, resp.) e times, and deleted e times, because UL and LL represent the active edges, and no edge becomes active twice, and at the end, no edge is active anymore. Thus, line (10) - (13) and (20) - (24) are executed e times. Line (14) performs the increment of k_2 and is executed for each crossing of situation 2. Since there is at least one instance of w in UL (because we check that *last_occurrence*(w) is not nil, thus at least *last_occurrence*(w) is in UL), we are sure that the value of k_2 is later added to k_3, that is finally added to *nr_crossings*. Thus, line (14) is executed not more than c times.
It remains to take a look on the implementation of

a downward layout, or if it creates an unintended horizontal edge to a node at the same level. The following algorithm sketches the fine tuning heuristics. As result of this fine tuning, all edges adjacent to a node get a balanced length with respect to their priorities:

while $\exists v \in V$ with $W_1(v) > threshold$ **do**
  $T = R(v) + \lfloor W_1(v) \rfloor$;
  if $feasible(T, v)$ **then** $R_{neu}(v) = T$; **fi**
  if *timeout* **then break**; **fi**
od

3.2 A n-Level Hierarchy

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 [BET93][War77].

Definition 5 *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 $E_i \subseteq V_i \times V_{i+1} \times A \times P \times C$.*

To construct a proper hierarchy from $G = (V, E)$, upward edges are reversed. Of course, they are marked such that arrowheads in the drawing will show the original direction. Furthermore, edges crossing several levels are split into small edges and dummy nodes. Finally, for all edges (v, w, a, p, c) holds $R(v) = R(w) - 1$. The following algorithm sketches the partitioning of edges.

procedure *check_edge* (e)
  let $e = (v, w, a, p, c)$;
  if $R(v) > R(w)$ **then**
    e_1 =*reverse*(e); *replace e by e_1*;
    check_edge(e_1);
  fi
  if $R(v) = R(w)$ **and** *v and w can be neighbored* **then** *fusion*(v, w);
  fi
  if $R(v) < R(w) - 1$ **or** $R(v) = R(w)$ **and** *v and w cannot be neighbored* **then**
    d =*create_dummy at level* $R(v) + 1$;
    replace e by $e_1 = (v, d, a, p, c)$ *and*
      $e_2 = (d, w, \_, p, c)$ *which are new edges*;
    check_edge(e_2);
  fi
end

4 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 [EMKW86], thus we use a heuristics.

4.1 Calculation of Crossings

In [STM81] and [War77], a representation of a proper n level hierarchy is proposed by using $n-1$ interconnection matrices of size $|V_i| \times |V_{i+1}|$. The representation of the levels is crucial, because in the following, we often calculate information of a level. Thus, we store each level V_i as a doubly linked linear list of nodes, and edges from E_i as predecessor and successor adjacency lists. This has three advantages: First, the interconnection matrix does not easily allow multiple edges between two nodes, which is possible in adjacency lists, and which occurs in our applications often. Secondly: doubly linked lists of nodes with adjacency lists require only a space of $O(|V_i| + |V_{i+1}| + |E_i|)$. Thirdly: The calculation of the number of crossings between two levels is given in [STM81] by a nested sum over the interconnection matrix, and thus needs time $O(|V_i|^2\ |V_{i+1}|^2)$. Instead, we use a sweep line algorithm of the complexity $O(|V_i| + |V_{i+1}| + |E_i| + c)$ where c is the number of crossings.

A similar plane sweep algorithm is described in [BeOt79], however for the more general case of arbitrary line segments in the plane with runtime $O((|E| + c)\ log|E|)$. But for the number of crossings between two levels of nodes, the situation is much simpler:

- The sequence of nodes in the doubly linked lists forms already the horizontal ordering of nodes, thus the line segments need not to be ordered, i.e. the lower bound $\Omega(|E|\, log|E|)$ given in [PrSh85] for the more general case does not hold here.
- Between two adjacent levels, all line segments go from the upper to the lower level. Their start and end points have the same vertical positions. Thus, the sweep line need not to be implemented by a dictionary (e.g. a balanced tree) reporting vertical positions of line segments. This allows to reduce the execution time by the factor $log|E|$.

The number of crossings depends only on the order of the nodes within the levels; it is independent of the relative position of the nodes of the upper

edge be drawn? We define as convention, that an arbitrary node of the subgraph is the target of the edge.

- If a 'region_fold' operation is performed before a 'graph_fold' operation, then it is unclear which subgraph the region summary node belongs to. We solved this by the convention that the summary node belongs to the same subgraph as an arbitrary one of the start nodes of the region.

- If a folded region or subgraph contains a start or end node of another 'region_fold' operation, then the latter operation cannot be executed correctly. In this case, we decide to inherits the property of the summary node to be start or end node from the nodes that are folded.

By these conventions, it is straightforward to calculate 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 Rank Assignment

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. The ranks correspond to the depth of the nodes in a spanning tree of the graph.

Definition 3 *Let $G = (V, E)$ a flat, connected graph. A tree $A = (V, T)$ with $T \subset E$ and $|T| = |V| - 1$ is called* **spanning tree** *of G. The rank $R(v)$ is the sum of the number of edges on a path from a root node of A to v in A.*

There are many possibilities to calculate the rank:

- Calculate a spanning tree by depth first search or breadth first search. This has the time complexity $O(|V| + |E|)$, but it results in an arbitrary partition [Meh84b].

- Calculate a minimum cost spanning tree $A = (V, T)$, i.e. with $\sum_{e \in T} C(e)$ is minimal, where C is the cost function on edges. Because the edges of the spanning tree are later drawn between adjacent levels, these edges are rather short. We define $C((s, t, a, p, c)) = 1/p$, which implies that edges of high priority are preferable part of the spanning tree, thus they are preferable short. To improve this behavior, we calculate the minimum cost spanning tree on the undirected graph that results canonically from the flat graph G. The complexity of this algorithm is $O(|E|\, log|V|)$ by reducing it to a union find problem. [ChTa76] present a better algorithm with complexity $O(|E|\, loglog|V|)$.

- If the graph is acyclic, then we can use topological sorting to calculate the rank. $R(v)$ is set to $max\{R(w) \mid w \text{ is predecessor of } v\} + 1$. This algorithm results in a downward directed layout and needs time $O(|V| + |E|)$.

Because nodes of the same rank are laid out at the same level, 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. Of course, this is done if only one neighbor node at the left or right side exist. During the partitioning, neighbored nodes are therefore fused into one node.

Definition 4 *v and w are* **neighbor nodes** *if they have the same rank $R(w) = R(v)$ and there is an edge (v, w, a, p, c) with a is* `left` *or* `right`*.*

This rank assignment allows to give a first estimate of the quality of the layout. Edges between adjacent levels are approximately shorter than edges between nonadjacent levels. In [GKNV93], an optimal ranking is formulated as an integer program:

$$min \sum_{(v,w,a,p,c) \in E} p\ (R(v) - R(w))$$

subject to $R(v) - R(w) > 0$ and is solved by a network simplex algorithm. In the VCG tool, the initial rank assignment is improved with respect to the same cost function by a fine tuning heuristics using the following node weight:

$$W_1(v) = \frac{W_2(v)}{\sum_{(w,v,a,p,c) \in E} p + \sum_{(v,w,a,p,c) \in E} p}$$

where

$$W_2(v) = \sum_{(v,w,a,p,c) \in E} p\ (R(w) - R(v)) + \sum_{(w,v,a,p,c) \in E} p\ (R(w) - R(v))$$

A new rank of a node is not feasible, if it is negative, if it creates an unintended upward edge in

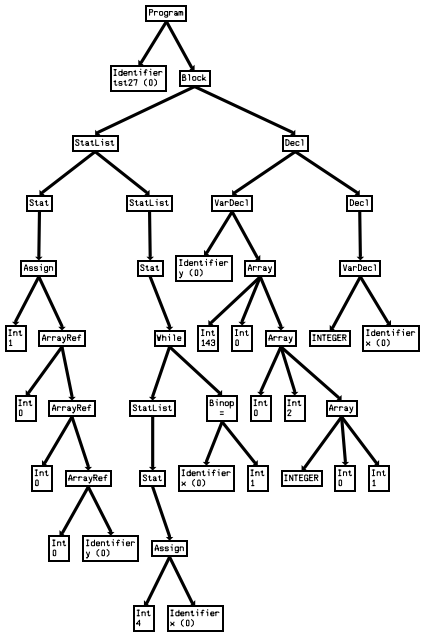


Figure 6: The same syntax tree: type annotations are hidden

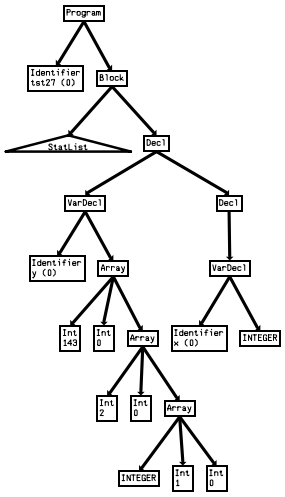


Figure 7: The same syntax tree: a subtree is folded

cent to these text lines (see figure 9). 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) 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.

Definition 2 *The* **layout problem** *of a graph* $G = (V, E, w, h)$ *and a sequence of operations 'fold_graph', 'hide_class', and 'fold_region'*

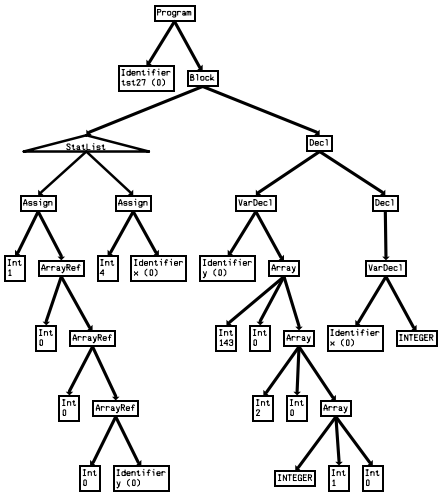


Figure 8: The same syntax tree: a region is folded

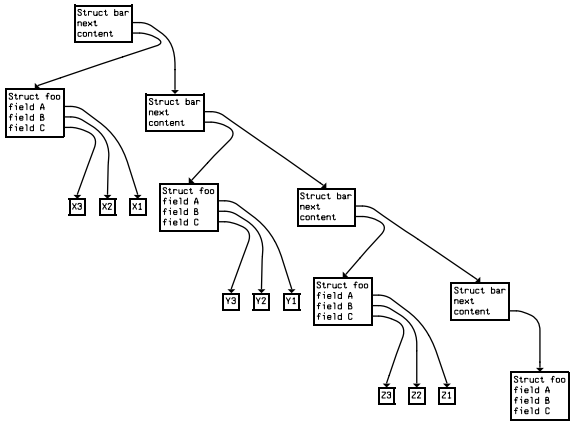


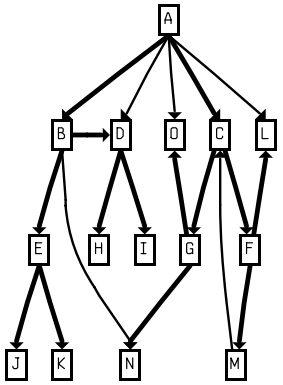
Figure 9: Visualization of Structs by Anchor Points

is to assign coordinates $(x(v), y(v))$ *to all visible nodes* $v \in V$ *and a sequence of polygon segments or Bezier splines with control points* $(x_0(e), y_0(e)), \ldots, (x_n(e), y_n(e))$ *to each visible edge* $e \in E$*, such that*

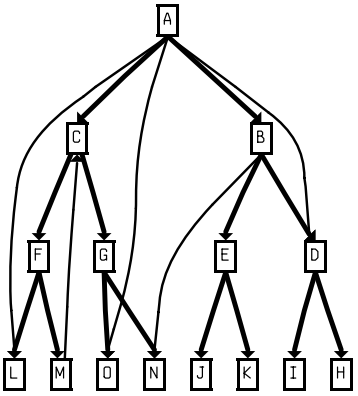
- *the visible nodes do not overlap,*
- *no edge goes through a visible node,*
- *each edge* (s, t, a, p, c) *starts at the corresponding anchor point* $a(s) = (x_0(e), y_0(e))$ *and end at an arbitrary anchor point* $(x_n(e), y_n(e))$ *of* t*.*

There are three technical difficulties in folding operations:

- If an edge points to a subgraph but the subgraph is not folded, to which node should the



All edges have the same priority.



The tree edges have the higher priority.

Figure 4: A Tree with Additional Edges

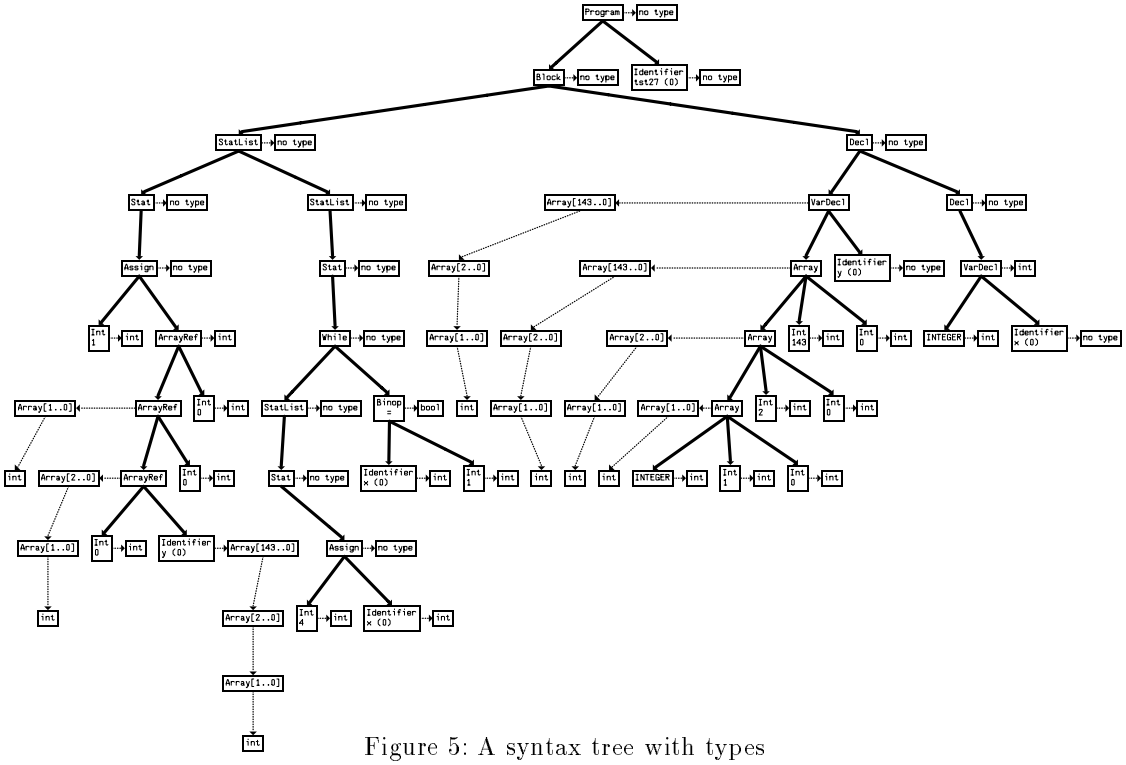


Figure 5: A syntax tree with types

The priority p of an edge (s, t, a, p, c) indicates the edge's importance. High priority numbers indicate high importance. This allows to layout graphs that are interwoven with some additional edges, where we want to recognize the structure of the base graph. For instance figure 4 shows a tree (thick edges) with some additional edges. Because all edges have the same priority, the tree structure is not recognizable on the left picture. By increasing the priority of the thick edges, the tree structure becomes visible on the right picture.

The anchor point a of an edge (s, t, a, p, c) 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 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 adja-

annotated by attributes:

Definition 1 *A* **nested graph** $G = (V, E, w, h)$ *is recursively defined: it consists of a set* V *of nodes and a set* $E \subseteq (V \times V \times A \times P \times C)$ *of edges, and a width* w *and height* h.
A node $v \in V$ *is either a simple node* (w, h) *with width* w *and height* h, *or a nested graph* G.
An edge (s, t, a, p, c) *consists of the source node* s, *a target node* t, *an anchor point* a, *a priority* p *and an edge class* c.
If a graph does not contain nested subgraphs, we call the graph **flat**.

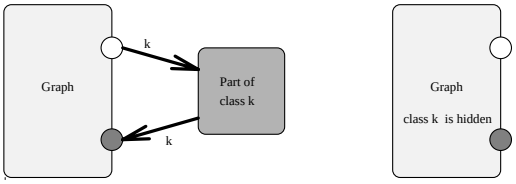


Figure 1: Hiding of Edges and their Region
The annotation (dark grey box) is a graph where all edges are in class k while the main graph is connected via class j ($j \neq k$). The bold edges of class k connect the annotation with the main graph, thus after hiding with respect to class k, the annotation is invisible. Other nodes (e.g., the grey node) in the main graph are unchanged even if there are edges from the invisible annotations to these nodes.

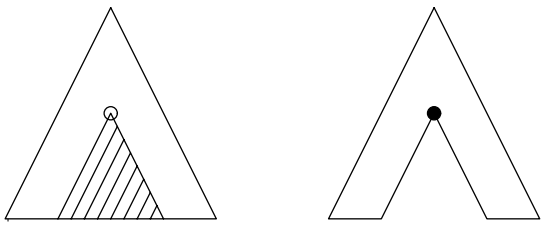


Figure 2: Folding a Subtree
The striped subtree is folded to the black summary node.

Graphs can be folded in different ways. The aim of a folding operation is to reduce the amount of the object 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:

- **graph_fold**: A subgraph (V, E, w, h) can be folded, i.e. all its nodes and edges disappear. Instead, a summary node is drawn. The width and height of the unfolded subgraph depends on the layout. The width and height of the summary node of the folded subgraph is specified by w and h.

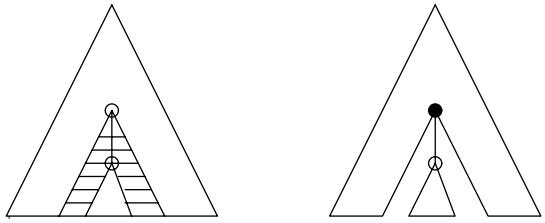


Figure 3: Folding a Part of a Subtree
The striped region is folded to the black summary node.

- **hide_class**: All edges of an edge class can be hidden. They disappear, but unfortunately, they often leave single nodes that are not anymore connected with the rest of the graph. It is appropriate to remove these single nodes, too.
- **region_fold**: While 'hide_class' changes the graph globally, a similar local operation can be used: 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.

The operation 'hide_class' allows to hide regions of the graph that are only connected by a certain class (see figure 1). Applications are hiding of annotation parts of trees or graphs. A simple example of the operation 'region_fold' is a tree where all edges have the same class k. Folding a node n with respect to the predicate '$\text{class}(e) = k$' is folding the whole subtree starting from n to one summary node (see figure 2). Folding n until m (where m is in the subtree of n) is folding the path from n to m and all subtrees along this path except the subtree that starts from m (see figure 3).

Figure 5 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_class 2' is shown in fig. 6: The type information has disappeared. In fig. 7, the region starting at the uppermost 'StatList' node is folded and represented by a triangle. In fig. 8, the region starts with the same node but end at the both 'Assign' nodes. In our example, we used the predicate 'e has edge class ≤ 2' for these operations 'fold_region'.

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) [AAS94] – 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 [WiMa92]. A simple textual visualization of trees and graphs is too confusing or even unreadable. A special visualization tool that draws trees and graphs in a natural way is more helpful.

The main problem in graph visualization is to place the nodes and edges of a graph in the plane such that the resulting layout makes the structure of the graph visible. Since the calculation of a nice layout is computationally hard [Bra90][GaJo83] 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.

Whether a graph looks nice, depends on the personal taste of the user. However, there are some common aesthetic criteria that are used by the most graph layout tools to find good placements of nodes:

- Place the nodes in a hierarchy of layers.
- Avoid crossings of edges and nodes.
- Keep edges short and straight.
- Favor a balanced placement.
- Position related nodes close together.

There may be graphs where some of these criteria are contradicting. For instance, to avoid that an edge is drawn through a node, it may be necessary that the edge is bent. In these cases, a compromise is necessary: The VCG tool always distributes the nodes in a hierarchy and avoids to draw an edge through a node, while symmetry and closeness of the layout are regarded as of minor importance.

An *interactive* tool has an additional criterion, which is the most important:

- Be reasonably fast.

It is much more annoying for a user to wait a long time after every keypress or mouse selection than to see a graph that is slightly ugly. Furthermore, the user is often not interested in an overall nice picture, but wants to inspect details of a graph. If the graph is large, the user does not see all nodes at the same time; thus the overall layout can be ugly, as long as the visible region is readable and the tool provides facilities to follow edges and find nodes. Henry in his Ph.D.Thesis [Hen92] describes the exploration of very large graphs and came to similar results.

In the following sections, we describe the layout algorithm of the VCG tool. This algorithm has common parts with the layout algorithms of similar tools, but in many cases, it is faster or can deal with larger graphs. A good overview of the literature of graph drawing is given in [BET93]. Related work is the development of the EDGE tool [PaTi90] which was used as predecessor of the VCG tool in the project COMPARE, and which has a nearly compatible specification language, and the tools daVinci [FrWe93], DAG [GKNV93] [GNV88], and DOT [KoEl91], which use similar approaches. The graph editor Graph^{Ed} [Him89] includes a large collection of algorithms to create, analyze and layout graphs interactively. Our work is based on the approach of Sugiyama, Tagawa, and Toda [STM81]. We extend their algorithm by several new heuristics, by avoiding their expensive matrix operations in order to be able to handle large graphs, by the possibility to include priorities and anchor points of edges in the layout algorithm, and by a simple spline routine for drawing edges.

In the next section, we give an introduction to how to influence the layout by the VCG tool. A requirement specification for the layout algorithm is derived. The following sections are organized according to the phases of the algorithm: after folding to the parts of the graph that must be laid out, the nodes are partitioned into layers according their rank. Next, the nodes are ordered within the layers to avoid crossings, then the nodes are placed to fixed coordinates. Finally, the bendings and splines of the edges are calculated. At last, we present some experiences and statistics concerning speed and applicability.

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,

Contents

Graph Layout through the VCG Tool

Georg Sander*
(sander@cs.uni-sb.de)

Technical Report **A03/94**
Universität des Saarlandes,
FB 14 Informatik,
66041 Saarbrücken

October 4, 1994

Abstract

The VCG tool allows the visualization of 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.

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