Graph Layout for Applications in Compiler Construction

Technical Report A/01/96

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

Universität des Saarlandes,
FB 14 Informatik,
66041 Saarbrücken

February 27, 1996

Abstract: We address graph visualization from the viewpoint of compiler construction. Most data structures in compilers are large, dense graphs such as annotated control flow graph, syntax trees, dependency graphs. Our main focus is the animation and interactive exploration of these graphs. Fast layout heuristics and powerful browsing methods are needed. We give a survey of layout heuristics for general directed and undirected graphs and present the browsing facilities that help to manage large structured graphs.

Contents

Abstract: We address graph visualization from the viewpoint of compiler construction. Most data structures in compilers are large, dense graphs such as annotated control flow graph, syntax trees, dependency graphs. Our main focus is the animation and interactive exploration of these graphs. Fast layout heuristics and powerful browsing methods are needed. We give a survey of layout heuristics for general directed and undirected graphs and present the browsing facilities that help to manage large structured graphs.

1 Introduction

We address graph visualization from the viewpoint of compiler construction. Drawings of compiler data structures such as syntax trees, control flow graphs, dependency graphs [WiMa95], are used for demonstration, debugging and documentation of compilers. In real world compiler applications, such drawings cannot be produced manually because the graphs are automatically generated, large, often dense, and seldom planar. Graph layout algorithms help to produce drawings automatically: they calculate positions of nodes and edges of the graph in the plane.

Our main focus is the animation and interactive exploration of compiler graphs. Thus, fast layout algorithms are required. Animations show the behaviour of an algorithm by a running sequence of drawings. Thus there is not much time to calculate a layout between two subsequent drawings. In interactive exploration, it is annoying if the user has to wait a long time for a layout. Here, a good layout quality is needed, but the speed of visualization is even more important. As long as the layout quality is good enough to comprehend the picture, the user may accept small aesthetic deficiencies of the drawing.

In contrast, consider graph visualization for textbook publishing. Here, typically pictures of small graphs are used to demonstrate idealized abstractions of facts. Such pictures are mostly produced by hand. Their quality must be optimal in order to make the facts very easily comprehensible for the reader of the textbook. If automatically calculated layout is used, the techniques are different from those in interactive visualization: The calculation time may range up to hours because the quality of the drawing is more important in textbook publishing.

Layout techniques for interactive graph exploration usually are iterative heuristics. Iterative algorithms allow to trade time for quality. If the layout

quality is not satisfactory, more iterations are calculated, which is slower but gives better results. Heuristics are used because this allows to satisfy several potentially contradicting aesthetic requirements in a balanced way. General aesthetic layout criteria include minimization of edge crossings and node overlappings, display of symmetries, reduction of bend points in edges, uniform orientations of directed edges, and closeness of related nodes.

Apart from the layout heuristics, powerful browsing mechanisms are needed for interactive graph exploration. Many facilities such as unlimited scaling, searching of nodes, and following chains of edges are offered as a matter of course in today's graph drawing tools. Some advanced facilities are grouping nodes, collapsing groups into summary nodes (folding), hiding classes of nodes, and displaying special views onto the graph.

We present layout methods and browsing facilities suitable for graph visualization in compiler construction. After defining the general notation, the section 3 gives a survey of straight line drawing heuristics derived from physical models. Section 4 presents variants of a method for layered (hierarchical) layouts. Section 5 sketches some ideas about interactive grouping and folding of graphs, and section 6 presents browsing facilities with special views. Most of the mentioned algorithms and methods are implemented in the VCG tool, a graph layout tool designed for applications from compiler construction [Sa94]. All examples of this paper are generated by the VCG tool.

2 Notation

A (directed) graph $G = (V, E)$ consists of a set V of nodes and a set E of ordered pairs of nodes. An element $(v, w) \in E$ is called an edge of the graph. A graph is *undirected* if for each edge $(v, w) \in E$ also $(w, v) \in E$ holds. The set $\mathbf{pred}(v) = \{w \in V \mid (w, v) \in E\}$ is the set of *predecessors* of a node $v \in V$. The set $\mathbf{succ}(v) = \{w \in V \mid (v, w) \in E\}$ is the set of all *successors* of a node v. The sizes of these sets are $\mathbf{indeg}(v) = |\mathbf{pred}(v)|$ and $\mathbf{outdeg}(v) = |\mathbf{succ}(v)|$. The degree of a node v is $\mathbf{degree}(v) = \mathbf{indeg}(v) + \mathbf{outdeg}(v)$.

A sequence $v_0, \ldots, v_n$ is a *path* from v_0 to v_n if there are edges $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq n$. A *cycle* is a nonempty path from v to v. A graph without cycles is called *acyclic*. A graph is *dense* if it contains many edges and *sparse* if it contains only few edges. It would be superfluously pedantic to define these notions precisely. A graph with $|E| \leq |V|$ is always considered sparse, while a graph with $|E| \approx |V|^2$ is always considered dense.

3 Force and Energy Controlled Placement

The simplest kind of graph layout is a straight line layout. All edges are drawn as straight lines between the centers of the adjacent nodes. Calculation of the layout reduces to the problem of finding node positions.

The main idea of the heuristic is to simulate physical-chemical models. Many objects occurring in physics and chemistry (e.g. molecules, crystals, combined inoperative pendulums, etc.) have a high degree of uniformity and balance. These are just the aesthetic criteria aimed at by a good layout method. The uniformity of physical-chemical objects is a result of the force and energy effects at the particles. The particles move according to the forces, and come to inoperative positions when the forces eliminate each other, and the physical system is balanced if the energy sum is minimal. In the heuristics, we consider the nodes as particles, start from an arbitrary initial position, simulate the movements of the nodes and lower the energy stepwise such that the nodes come to rest.

(1) *Set all nodes $v \in V$ to initial positions;*
(2) **for** *actround = 1* **to** *maxrounds* **do**
(3) *Select a node $v \in V$;*
(4) *Calculate the forces at v;*
(5) *Move v an amount δ into the direction of the sum of forces;*
(6) *Calculate the energy E of the system;*
(7) **if** *E is small enough* **then stop loop**;
(8) **od**

3.1 Spring Embedding

The earliest heuristics of force-directed placement were based on the spring embedder model [QuBr79, Ea84]. Nodes are considered as mutually repulsive charges and edges as springs that attract connected nodes.

Let $\Delta(v,w)$ be the distance vector between two nodes v and w. Then, $\|\Delta(v,w)\|$ is the Euclidean distance. Between each pair of nodes, there are repulsive forces inversely proportional to the distance, e.g. the force vector

$$F_{rep}(v,w) = -\lambda_{rep}\,\frac{\Delta(v,w)}{\|\Delta(v,w)\|^2}$$

Between nodes connected by edges (v,w), there are attractive forces directly proportional to (a power of) the distance, e.g.

$$F_{att}(v,w) = \lambda_{att}\,\Delta(v,w)\|\Delta(v,w)\|^2$$

Different formulas for forces have been used in [QuBr79, Ea84, SuMi94, SuMi95], but the resulting effects are always similar. The parameters λ_{rep} and λ_{att} allow to adapt the heuristics. An edge (v,w) is at equilibrium if $F_{rep}(v,w) +$

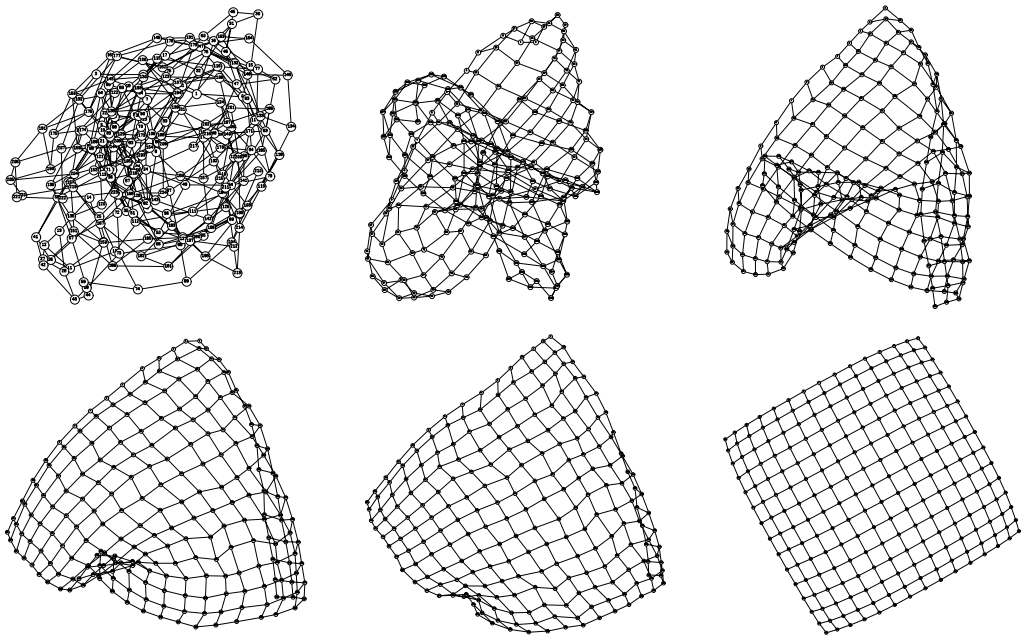


Figure 1: Animation of Spring Embedding of Grid Graph

$F_{att}(v,w) = 0$. The length of the edge in this case is

$$\|\Delta(v,w)\| = \sqrt[4]{\frac{\lambda_{rep}}{\lambda_{att}}}$$

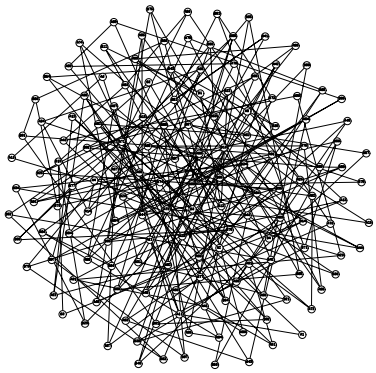
Although the algorithm does not explicitly support the detection of symmetries, it turns out that in many cases the resulting layout shows existing symmetries. If the iteration steps are animated, there is the impression of a three-dimensional unfolding process starting with a randomly produced bunch. The more symmetric a graph is, the more obvious is this effect. Fig. 1 shows the animation sequence of a regular grid graph.

3.2 Gravity

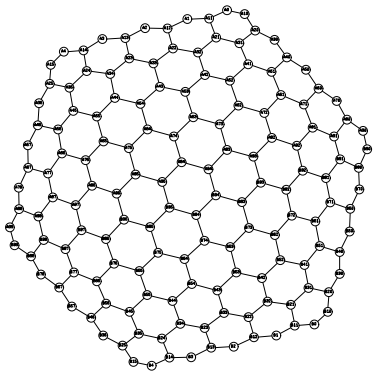
It is obvious that connected components of a disconnected graph will move apart in a simple spring model because of lack of attractive forces. Often, loosely connected components are also positioned far from each other such that the edges in between are unaesthetically long. Thus, Frick e.a. introduce additional gravity forces [FLM95]. All nodes $v_1, \ldots, v_n$ are attracted by the gravity to the barycenter (the average of all node positions $p(v)$):

$$B_{center} = \frac{1}{n} \sum_{i=1}^{n} p(v_i)$$

In the proposal of Frick e.a., gravity forces depend on the number $\mathbf{degree}(v)$ of adjacent edges at a node v. Nodes with high degree are more important since



Gravity and charge repulsion, without attractive spring forces



Gravity, repulsion and attractive spring forces

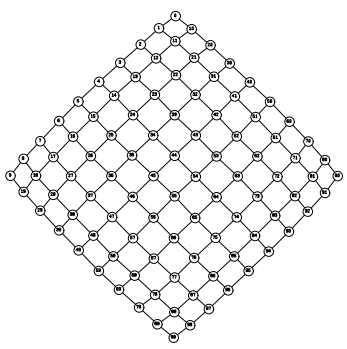
Figure 2: Layout of Hexagonal Grid

they drag along many nodes in the same direction. The gravity force at a node can be defined as

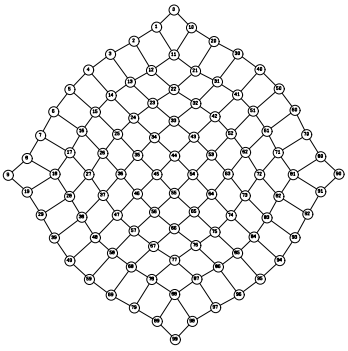
$$F_{grav}(v) = \lambda_{grav}(1 + \mathbf{degree}(v))(B_{center} - p(v))$$

Although gravity forces are attractive as of themselves, they are not a total replacement of spring forces. If only gravity and charge repulsion take effect, the nodes are placed evenly around the barycenter, but regularities of the edge structure are not visible (Fig. 2, left). Only the spring forces contribute to the symmetry of the layout.

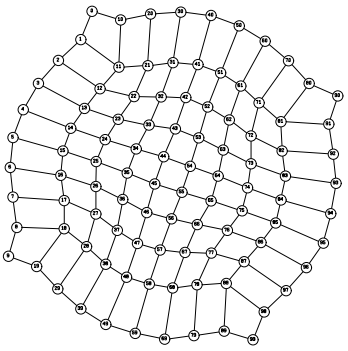
Since gravity forces are polar directed to the barycenter, they enforce a round structure of the layout. Fig. 3 shows the effect of gravity on a grid graph. However, the main advantage of gravity is visible if the graph is partitioned into very dense parts which are loosely connected. Without gravity, the nodes of the parts are very close together but the parts themselves are far from each other. Thus, the edge lengths are not uniform. Gravity has the effect that the



without gravity

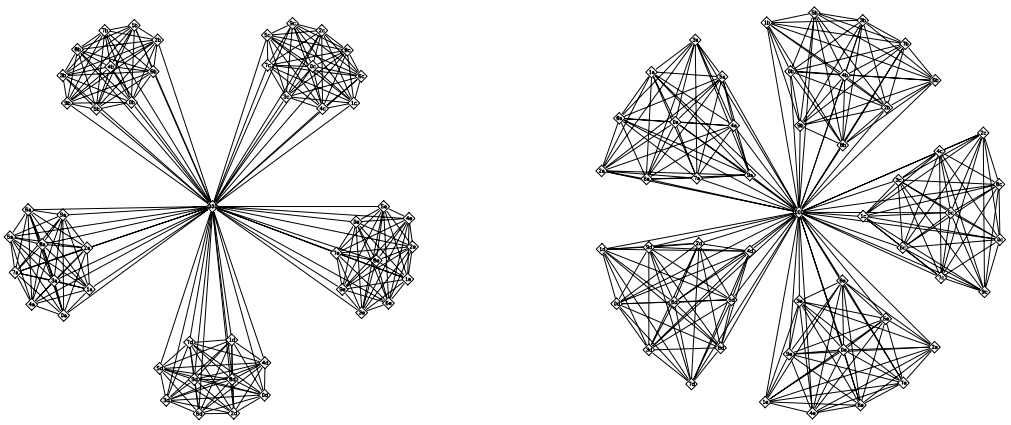


gravity $\lambda_{grav} = 0.6$



strong gravity $\lambda_{grav} = 1.5$

Figure 3: Layout with Gravity



without gravity

strong gravity $\lambda_{grav} = 2.0$

Figure 4: Layout of Multiplied K_{10}

parts are positioned closer such that the layout is much more homogeneous (Fig. 4).

3.3 Magnetic Fields

Spring embedders do not take into account edge directions. In directed graphs, all edges should point into the same direction when possible. Recently, Misue and Sugiyama [SuMi94, SuMi95] proposed an extension that enforces this effect: Edges are considered as springs, but also as magnetic needles which are oriented according to a magnetic field. Spring forces depend on the length of the edges and are parallel to the edges. A magnetic force additionally depends of the angle α between edge and magnetic field, and is directed orthogonally to the edge. Thus, it rotates the edge. The magnetic force becomes zero when the edge points exactly in the direction of the field (Fig. 5). In the formula of magnetic forces, $\perp(v, w)$ denotes the unit vector orthogonal to $\Delta(v, w)$ and

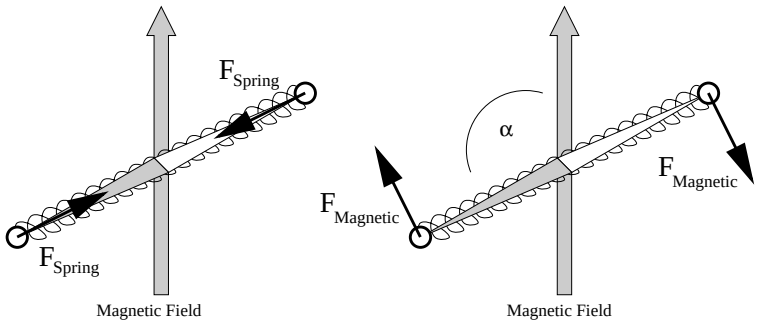


Figure 5: Spring Force and Magnetic Force

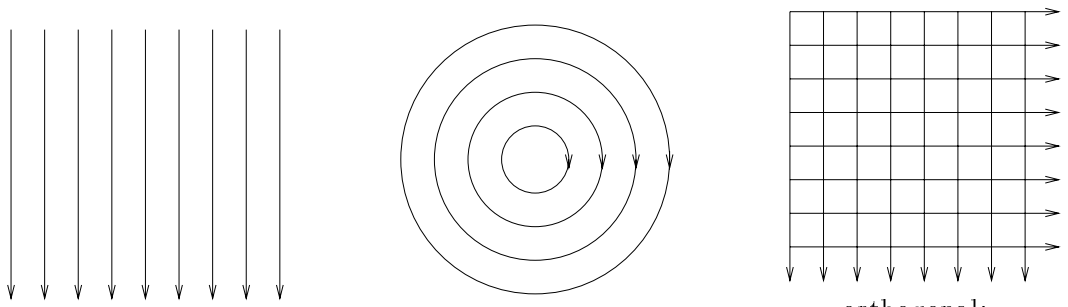


Figure 6: Magnetic Fields

the parameters λ_{mag} and c allow to tune the force:

$$F_{mag} = \lambda_{mag} \alpha^c \|\Delta(v,w)\|^2 \perp(v,w)$$

Different magnetic fields have been used (Fig. 6). A parallel field can be used to give most edges a top down orientation (Fig. 7). The number of edges pointing against the field direction depends on the strength of the field; it is small but seldom minimal.

A concentric field can be used to illustrate cycles in the graph (Fig. 8). Binary trees are often drawn in orthogonal layouts. A similar effect can be produced by a compound magnetic field where different sets of edges are influenced by different components of the field (Fig. 9). However, larger trees often produce edge crossings in the orthogonal field, such that this method is not perfectly suited for orthogonal drawings.

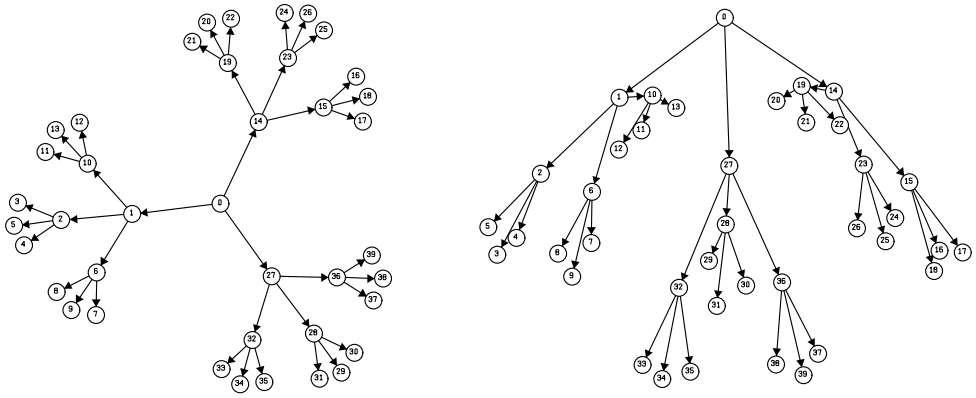
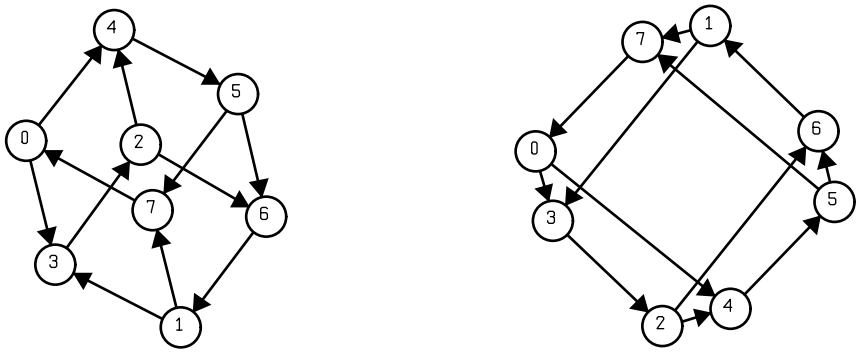
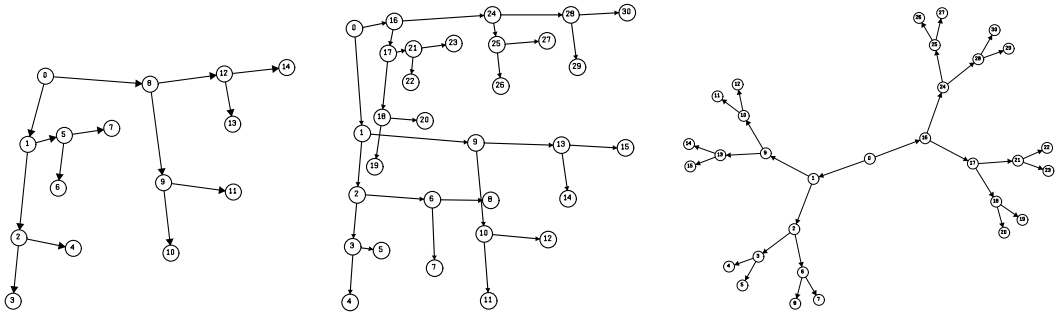


Figure 7: Ternary Tree with Magnetic Field



without magnetic field concentric field

Figure 8: Layout of Cube with Magnetic Field



orthogonal field orthogonal field without magnetic field

Figure 9: Layout of Binary Trees with Magnetic Fields

3.4 Simulated Annealing

The spring embedders of Eades [Ea84] and Misue und Sugiyama [SuMi94, SuMi95] apply a fixed number of iterations to get the layout. It may happen that the number of iterations is too small, which gives an unbalanced layout, or the number is too high, which is waste of time. Different extensions were proposed to get better termination conditions for the heuristic. Some spring embedders [QuBr79, KaKa89] are based on the energetic states of the nodes. The aim is to minimize the global energy E (the sum of all energetic states). A minimum of E can be found by solving the equation system

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_i} = 0 \text{ for } 1 \leq i \leq n$$

for the positions (x_i, y_i) of all n nodes. The equation system can be solved by numerical methods (e.g. the method of Newton-Raphson [BoPr91]). However, this only finds some local minimum of E which is not the global one.

Thus, Davidson and Harel [DaHa89] applied a randomized optimization method from statistical mechanics: simulated annealing. In addition to the global energy E, there is a global temperature T which is lowered as the

iterations progress. In each step, a random move is tried at some node. If the global energy E gets smaller with the new position of the node, the move is done. If E is enlarged by ΔE, the move is accepted with the probability

$$Prob = e^{-\frac{\Delta E}{T}}$$

otherwise the move is rejected.[1] The uphill changes of the energy prevent the layout to go towards a local minimum very early. By lowering the temperature T in each step, uphill changes get more improbable as the algorithm progresses.

As long as the temperature is decreased slowly enough, this randomized method results in uniform and symmetric layouts. The method has the advantage that no vector calculations are needed, because no force vectors need to be calculated. Any complex scalar formula for the energy is allowed, e.g. taking into account the border of the layout $x_{min}, x_{max}, y_{min}$ and y_{max}, or the number of crossings and overlappings. Typical formulas are

$$E_{global} = \sum_{\substack{v,w \in V \\ v \neq w}} E_{rep}(v,w) + \sum_{(v,w) \in E} E_{att}(v,w) + \sum_{v \in V} E_{border}(v) + E_{lap} + E_{cross}$$

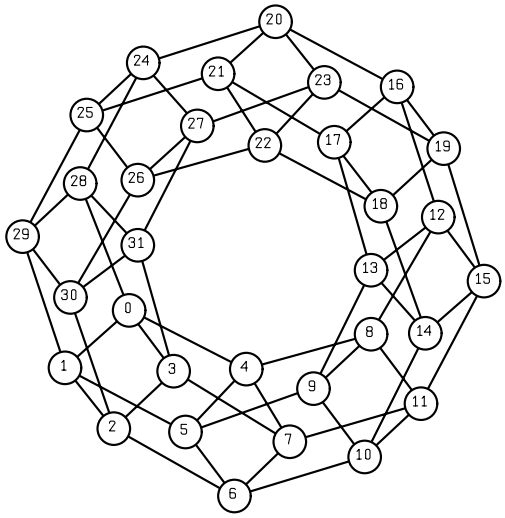
where

$$\begin{aligned}
E_{rep}(v,w) &= \frac{\lambda_{rep}}{\|\Delta(v,w)\|^2} \\
E_{att}(v,w) &= \lambda_{att} \, \|\Delta(v,w)\|^2 \\
E_{border}(v) &= \frac{\lambda_{border}}{(x(v)-x_{min})^2} + \frac{\lambda_{border}}{(x(v)-x_{max})^2} \\
&+ \frac{\lambda_{border}}{(y(v)-y_{min})^2} + \frac{\lambda_{border}}{(y(v)-y_{max})^2} \\
E_{lap} &= \lambda_{lap} \; \#Overlappings \\
E_{cross} &= \lambda_{cross} \; \#Crossings
\end{aligned}$$

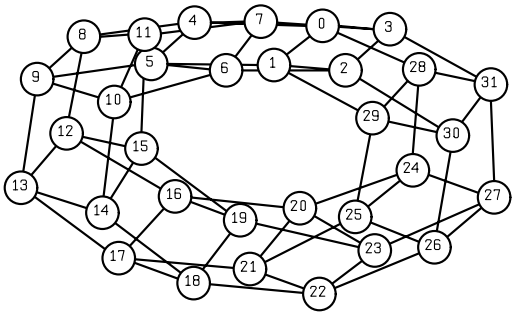
The disadvantage of simulated annealing is the fact that the cooling must be very slow to enforce regularities of the layout. It needs about 10 times more iterations than normal spring embedders (see also [BHR96] for a comparison between spring embedders and simulated annealing). Thus, it is not very well suited for large graphs.

Experiments have shown that the combination of both spring embedding and simulated annealing can be useful. We move the nodes in direction of the forces, but add a small random force F_{rand} and with a certain probability, we reject moves that would increase the global energy. Because the positioning of the nodes is not completely random, simulated annealing becomes faster, and

[1] This is derived from the Boltzmann probability of thermodynamic moves of particles of energy E in an ideal gas: $Prob = e^{-\frac{E}{KT}}$. Here, K is the Boltzmann constant.



Spring Embedder



Combined Method

Figure 10: Layout of Torus

because the energy state is checked, it is possible to enforce aesthetics that are not expressible as force vector. Fig. 10 shows an example: The spring embedder produces a symmetric layout of the torus, while the combined method allows to press the torus into a rectangular border.

3.5 Temperature Schemes

Fruchterman and Reingold [FrRe91] adapted the concept of cooling to the normal spring embedders. Nodes are moved in direction of the force vector instead of randomly. The amount of movement δ is limited by the global temperature T, i.e. the smaller T is, the smaller is the movement distance $\delta(T)$. If $T = 0$, the nodes don't move anymore. The global cooling function depends only on the size of the graph.

Frick e.a. [FLM95] expanded this concept by introducing local temperatures $T(v)$ for each node. The distance of movement of a node v is $\delta(T(v))$. Thus, the amount of movement may vary for each node. The global temperature is the average of all local temperatures: $T_{glob} = \frac{1}{n} \sum_{i=1}^{n} T(v_i)$. The simulation is iterated until T_{glob} is cooled down to a threshold T_{thresh}. There is no global cooling function, but the temperature of a node is determined by its movement behavior. The old movement impulse vector $I_{old}(v)$ is compared to the new impulse $I_{new}(v)$ (Fig. 11). If both nearly point into the same direction, the ideal position of node v can probably be found in this direction. Thus, its local temperature $T(v)$ is enlarged in order to move the node v faster (i.e. in larger steps) to its ideal position. If both nearly point into opposite directions, we assume that the node v was moved too much and now starts to oscillate around the ideal point. Thus, $T(v)$ is decreased to damp the oscillation until

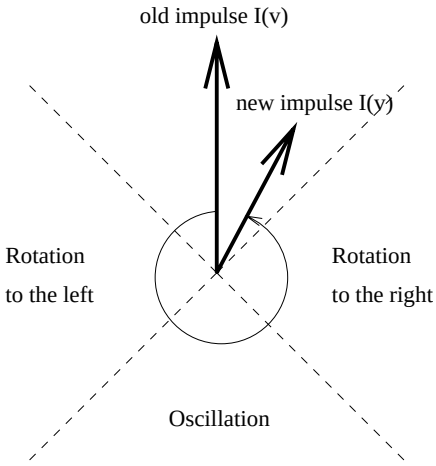


Figure 11: Detection of Rotations and Oscillations

the node has found its inoperative position. If a node turns several times in the same direction to the right (or to the left, respectively), it probably circles around its ideal point (like a rotation), thus $T(v)$ is decreased, too. Since the local temperature is sensitive to the movement behavior, it is automatically recognized when the simulation can stop without wasting iterations where the layout quality does not change anymore.

3.6 Applications in Compiler Construction

Although magnetic fields can be used to influence orientations of directed edges, force-directed and energy-controlled placement is mainly used for *sparse, undirected* graphs, e.g. symmetric relations. A typical example is the visualization of register collision graphs in compiler construction. If a compiler translates a program into machine code, it uses an infinite set of virtual processor registers at first. This is for simplicity of the code generation. Afterwards the virtual registers must be mapped to the limited number of real CPU registers. Here, the so called register collision graph helps: The nodes of this graph are the virtual registers. There is an (undirected) edge between two nodes if the life times of both virtual registers are overlapping. Register allocation is now done by coloring the graph with n colors representing n real CPU registers with the restraint that adjacent nodes must never get the same color. The problem of minimizing the number of colors (or the number of real CPU registers, resp.) is $\mathcal{NP}$-complete, but there are good heuristics to solve this problem [WiMa95].

Example 1 *Fig. 12 shows the register collision graph of the 3-address code of sequence 1. Here, the 8 virtual registers R1, ..., R8 are used. The graph is labeled by the life times of the registers. The graph can be colored by 4 real registers: R2 and R6 are mapped to the CPU register A, R2, R4 and R6 to B,*

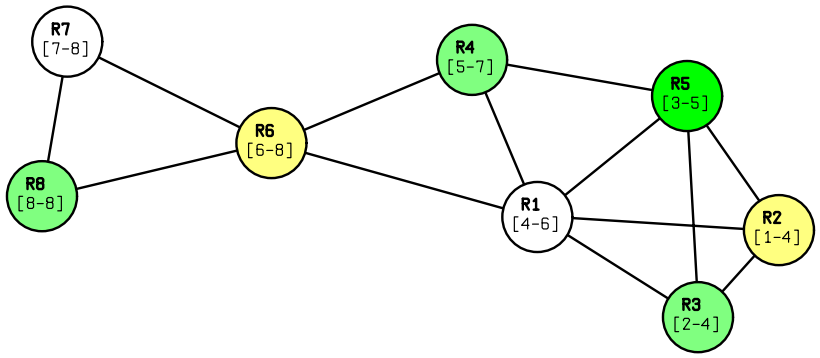


Figure 12: Register Collision Graph

R5 to C and R1 and R7 to D. The result is shown in sequence 2.

Sequence 1 (before reg. allocation): intermediate code using 8 virtual registers

```
(1) R2 = 1
(2) R3 = 7
(3) R5 = 9
(4) R1 = R2 + R3
(5) R4 = R1 * R5
(6) R6 = R1 + R4
(7) R7 = R6 * R4
(8) R8 = R7 + R6
```

Sequence 2 (after reg. allocation): code with 4 real registers

```
(1) A = 1
(2) B = 7
(3) C = 9
(4) D = A + B
(5) B = D * C
(6) A = D + B
(7) D = A * B
(8) B = D + A
```

Simulations of parallel programs often require the visualization of the parallel computer architecture. Because spring embedders often display symmetries, they are in particularly suitable for that. Fig. 13 shows some of these networks (see [AlGo89, Br93] for a description of these network topologies).

3.7 Related Approaches

Genetic layout algorithms [Ma92, Pa89] are very similar to simulated annealing. They are randomized methods that calculate generations of layouts of the same graph. The best layout (according to some quality function similar to the energy function of simulated annealing) is selected as new layout. Generations of layouts are produced by two operations in correspondence to biology: mutations (a layout changes randomly) and crossovers (two layouts are combined into a new layout). After a sequence of mutations and crossovers, the quality function is applied to all layouts. Bad layouts are deleted, and only the best layouts survive. Just as simulated annealing, genetic layout algorithms are relatively slow and need a lot of memory space.

Tunkelang [Tu94] developed a method similar to simulated annealing which

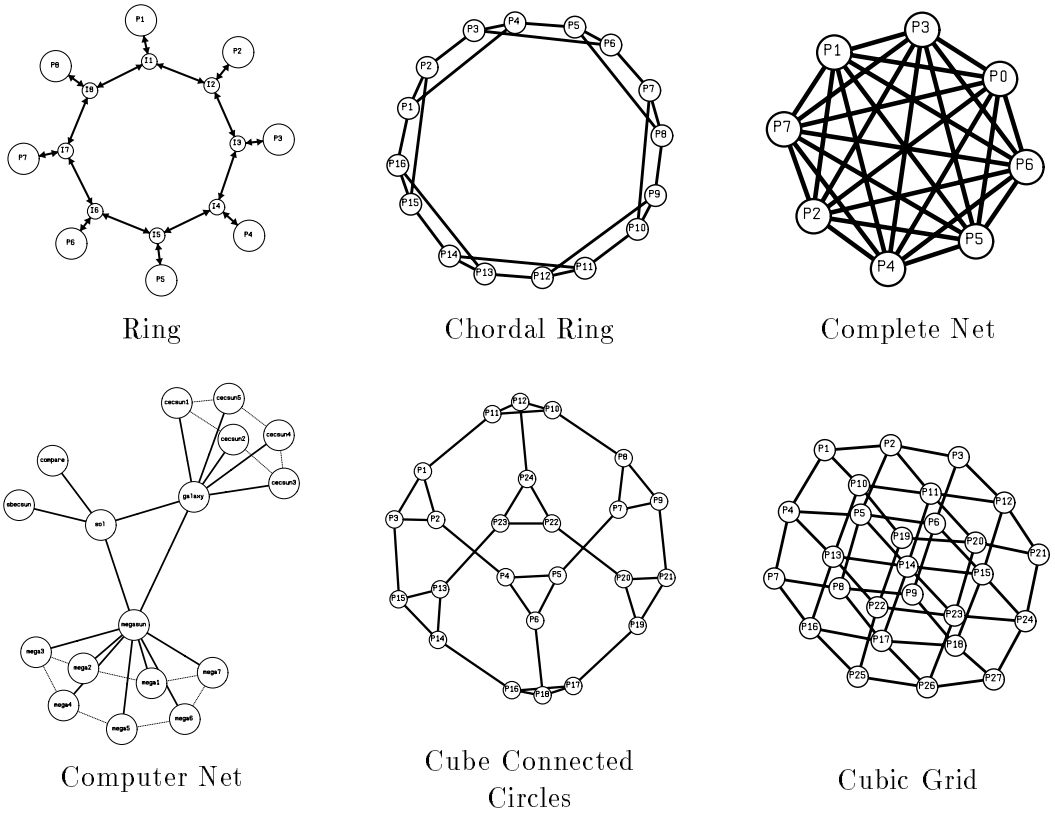


Figure 13: Network Topologies

does not place the nodes randomly but according to a fixed pattern. The energy function is applied during the initialization in order to find a good initial layout, and afterwards to improve the layout. The disadvantage of this method: with a fixed pattern of node movements some layouts are never taken into account. Thus, the algorithm does not always give optimal results. On the other hand, a good selection of the movement pattern may speed up the heuristics very much.

Spring embedders and simulated annealing do not necessarily produce planar layouts for planar graphs. Harel and Sardas [HaSa93] use a combined approach: First a planar layout (or for nonplanar graphs: a layout with only few edge crossings) is calculated, and afterwards, simulated annealing is used to improve the layout. In this optimization step, all node moves are rejected that would produce edge crossings. This guarantees symmetric, uniform planar layouts for planar graphs.

4 Layout in Layers

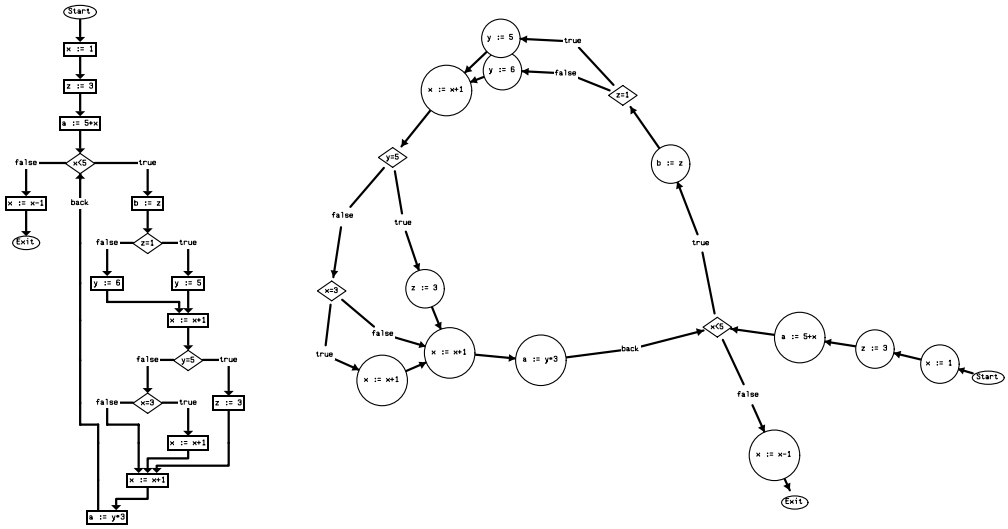
Straight line layout is sometimes not very useful for several reasons: (a) it does not ensure that nodes do not overlap, (b) is does not ensure that edges do not cross nodes, (c) it is for certain applications simply a wrong layout pattern. For instance control flow diagrams in compiler construction look completely different from typical straight line layouts. It is important that nodes do not overlap because their labels must be readable. In branches of the control flow, the user expects labels directly near the node that represents the branch condition. The start node of the control flow should be at the top. To draw such graphs differently may also produce nice pictures (see Fig. 14, right), but they look unfamiliar for users that expect a control flow graphs, because they do not satisfy the drawing conventions.

Next, we present a layout method that avoids node overlappings and allows edges with bends. Here, not only node positions must be found, but edge routing must be done, too.

4.1 Layout Phases

The main idea of the algorithm is to partition the nodes into layers and order the nodes within the layers such that edge crossings are reduced. Variants of this idea were first described in [Wa77, Ca80, STT81]. The method described here is mainly based on the algorithm by Sugiyama e.a. [STT81, EaSu90]. Layer layout consists of four phases (Fig. 15):

- Partitioning of nodes into layers. The goal is to construct a *proper hi-*



typical flow diagram straight line layout of this graph

Figure 14: Control Flow Graphs

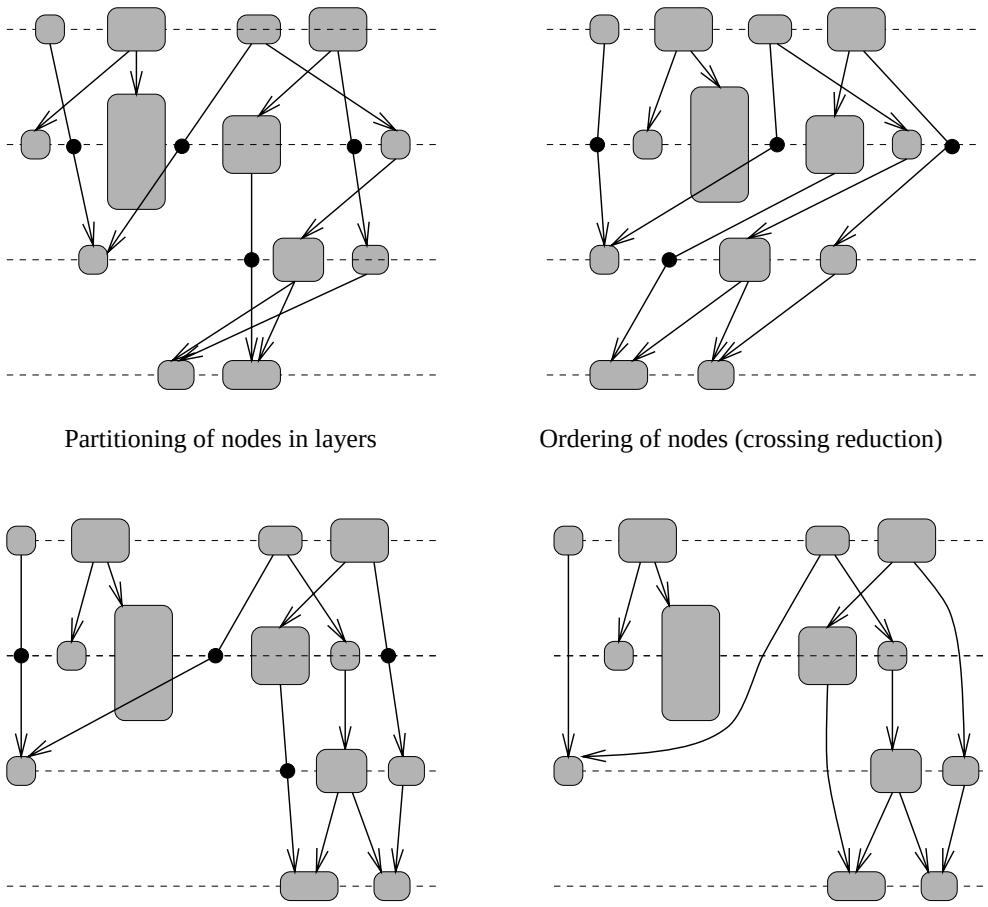


Figure 15: Phases of Layer Layout Algorithm

erarchy, i.e. a partioning where edges may only occur between adjacent layers. If this is not possible, long edges crossing several layers must be split into sequences of short edges and dummy nodes must be inserted appropriately.

- Sorting the nodes (and dummy nodes) within a layer, such that only few edge crossings exist. This gives the relative positions of the nodes.
- Positioning of nodes. This gives the absolute coordinates of the nodes. The goal is to find balanced positions without overlappings.
- Positioning of edges. Start and end points of edges are approximately given by the node positions, because they must be adjacent to the borders of the nodes. However, bend points must be calculated to avoid crossings through nodes, or control points for certain edge styles (e.g. splines).

4.1.1 Phase 1: Partitioning into layers

For each node v, a rank $R(v)$ has to be calculated, that specifies the number of the layer that v belongs to. Layer 1 is the topmost layer. The span of an edge is $S(v,w) = R(w) - R(v)$. In a directed graph, it might be required that most edges point downwards, i.e. that the spans are positive. However, it is $\mathcal{NP}$-complete to find the minimal number of edges that cannot point downwards in a graph which contains cycles [GaJo79]. There are many heuristics for calculation of the rank (some more are described in [EaSu90]):

- If the graph is acyclic, sort it topologically and calculate

  $$R(v) = \max\{R(w) \mid w \in \mathbf{pred}(v)\} + 1$$

  in topological order of the nodes. This results in a partitioning where all edges will point downwards in time complexity $O(|V| + |E|)$.

- If the graph is acyclic, solve the problem to minimize

  $$D = \sum_{(v,w)\in E} (R(v) - R(w) - 1)$$

  subject to (1) $R(v) \geq 1$ for each node v and (2) $R(v) - R(w) \geq 1$ for each edge (v,w). This can be done by standard linear programming methods [GKN93]. Even an integer solution exists and can be obtained. This results in a downward partitioning with minimal number of edge spans, i.e. minimal number D of dummy nodes.

- Calculate $R(v)$ by a depth first search or breath first search. This results in an arbitrary partitioning in time complexity $O(|V| + |E|)$.

- Calculate the minimum cost spanning tree [Me84] on the undirected instance of the graph. This is useful if edges e have priorities $p(e)$. We use the cost $1/p(e)$. The result is a partitioning where edges of high priority gave small spans. The layout will be wide but not deep. The time complexity is $O(|E| \log |V|)$.

- If the edge orientation is not important, apply a spring embedder as described in the section before. It is sufficient to take only F_{rep} and F_{att} into account. Instead of two-dimensional coordinates, calculate a one-dimensional coordinate $R(v)$. This results in a ranking where edges tend to have the same absolute value of span.

As mentioned above, some heuristics can only cope with acyclic graphs. Graphs with cycles have to be made acyclic first by (conceptually) reversing some edges. A heuristic to find these edges works as follows: Calculate the strongly connected components of the graph [Me84] in time $O(|V| + |E|)$. In

each component C that contains more than one node, reverse an edge. Now try again to calculate the strongly connected components. Continue this loop, until each component has only one element. At the end, the converted graph will be acyclic. A good heuristic to find the edges to be reversed is to look for edges (v, w) where **outdeg**(v) is minimal but **indeg**(v) and **indeg**(w) are maximal.

This method can be implemented by recursion. In practice, it very often finds the minimal number of edges that must be reversed, although it is only a heuristic. However, it has the high time complexity $O(r(|V| + |E|))$ where r is the number of reversed edges.

Each ranking induces a *hierarchy*. In order to proceed, a *proper hierarchy* is needed, i.e. all edges must have span $S(e) = 1$. Thus, edges (v, w) with $S(v, w) < 0$ are reversed, i.e. replaced by edges (w, v). Then edges with $S(v, w) = n > 1$ are split into dummy nodes $v_1, \ldots, v_{n-1}$ with $R(v_i) = R(v) + i$ and smaller edges $(v, v_1), \ldots, (v_i, v_{i+1}), \ldots, (v_{n-1}, w)$, and edges with $S(v, w) = 0$ are diverted in a similar way. Edge splittings and reversions are always done only conceptually. The resulting edges are marked such that we can later draw one arrowheads at the appropriate position.

4.1.2 Phase 2: Sorting of nodes

For each node v, a relative position $P(v)$ within its layer has to be calculated, such that there are only few edge crossings. Since the hierarchy is proper, the number of crossings c originated by the edges E_i between two adjacent layers V_i and V_{i+1} can be easily determined by a plane sweep algorithm [Sa94] in time $O(|V_i| + |V_{i+1}| + |E_i| + c)$. However, the problem of finding permutations of the sequences V_1 and V_2 to get a minimal number of crossings is $\mathcal{NP}$-complete [GaJo83]. Methods to solve the crossing problem can be found in [STT81, EaWo86, EaKe86, EaSu90, Sa94, JuMu96]. In practice, the most successful algorithm is the layer-by-layer-sweep:

```
(1)  while the crossing number is not satisfactory do
(2)      for each layer V_i from i = 1 to n do
(3)          for each v ∈ V_i do
(4)              Calculate weight W_p(v);
(5)          od
(6)          Sort the nodes of V_i according to the weight W_p(v);
(7)      od
(8)      for each V_i from i = n to 1 do ... similar with W_s(v) od
(9)  od
```

The first traversal (line (2)-(7)) is a top down traversal, the second (line (8)) is a bottom up traversal. Other variations of this method sweep only top down

or only bottom up, or from the center outwards. [Sa94] describes a variation with limited backtracking: if a sweep did not reduce the number of crossings, the old configuration is taken. The crucial point is the selection of the weights W_p and W_s. [STT81] proposes the barycenter weight ($P(w)$ is the relative position of the node w in the predecessor or successor layer, respectively):

$$W_p^{(b)}(v) = \frac{1}{\mathbf{indeg}(v)} \sum_{w \in \mathbf{pred}(v)} P(w)$$

$$W_s^{(b)}(v) = \frac{1}{\mathbf{outdeg}(v)} \sum_{w \in \mathbf{succ}(v)} P(w)$$

[EaWo86, GKN93] propose the median of the sequence $w_1, \ldots, w_{\mathbf{indeg}(v)}$ of predecessors of a node v:

$$W_p^{(m)}(v) = \frac{1}{2}(P(w_{\lfloor \frac{\mathbf{indeg}(v)}{2} \rfloor + 1}) + P(w_{\lceil \frac{\mathbf{indeg}(v)}{2} \rceil}))$$

We also made experiments with combinations of both, using

$$W_p^{(h)}(v) = \lambda_1 W_p^{(b)}(v) + \lambda_2 W_p^{(m)}(v)$$

A method of calculating the optimal permutation of two layers where one layer is fixed was proposed in [JuMu96]. Assume that the permutation of V_1 is fixed, and a permutation of V_2 should be calculated. Let c_{ij} denote the number of crossings among edges adjacent to $v_i, v_j \in V_2$ in a permutation of V_2 where $P(v_i) < P(v_j)$. Let $x_{ij} = 1$ if $P(v_i) < P(v_j)$, and $x_{ij} = 0$ otherwise. Then the number of crossings of a permutation of V_2 can be described as

$$C = \sum_{i=1}^{|V_2|-1} \sum_{j=i+1}^{|V_2|} (c_{ij} x_{ij} + c_{ji}(1 - x_{ji}))$$

The optimal permutation of V_2 can be found by calculating $x_{ij} \in \{0, 1\}$ such that C is minimal, subject to (1) $0 \leq x_{ij} + x_{jk} - x_{ik} \leq 1$ for $1 \leq i < j < k \leq |V_2|$, and (2) $0 \leq x_{ij} \leq 1$ for $1 \leq i < j \leq |V_2|$. The "3-cycle constraints" (1) guarantee that the result describes a valid permutation. This linear integer programming problem can be solved by a variation of the branch and cut algorithm [JuMu96]. This method is suitable up to $|V_2| < 60$, but it is much too slow for larger graphs.

Statistical experiments [JuMu96, Sa96b] show that apart from the optimal method for two layers where one is fixed, the best heuristic is $W^{(h)}$ with $\lambda_1 \gg \lambda_2$, followed by $W^{(b)}$, and at last by $W^{(m)}$. These methods are also closer to the optimum and faster than various greedy or stochastic methods described in [EaKe86, JuMu96]. However, this experimental result does not hold if there are more than two layers, and a layer-by-layer-sweep is used. Firstly, a sweep with the two-layer-optimal algorithm does not calculate the optimal crossing

number of the whole multi-layer-graph since a nonoptimal permutation of some adjacent layers might produce less crossings than a situation where the first layer is optimal, but the other layers are only *optimal derived* from the first layer. Secondly, it is not obvious which of $W^{(b)}, W^{(m)}$ and $W^{(h)}$ produces the fewest crossings in a multi-layer-graph: there are many examples where any of the three is the best.

4.1.3 Phase 3: Positioning of nodes

For each node v, absolute coordinates $X(v)$ and $Y(v)$ must be calculated such that (1) $R(v) < R(w)$ implies $Y(v) < Y(w)$ and (2) $P(v) < P(w)$ implies $X(v) < X(w)$. Nodes should not overlap. The layout should be balanced.

Again, we use a layer-by-layer-sweep that is motivated by physical models. As we have seen in section 3, physical simulations often result in very balanced positionings. We start with an arbitrary layout that satisfies conditions (1) and (2). The goal is to minimize

$$Z = \sum_{v \in V} | \sum_{\substack{(v,w) \in E \\ (w,v) \in E}} (X(w) - X(v))|$$

subject to condition (2) and to the condition that nodes must not overlap. Again, this could be solved by standard linear programming methods.

However, a heuristics that is much faster in practice is the rubber band network simulation: the edges pull the nodes like rubber bands. The nodes move horizontally according to the sum of the forces. We define the force

$$F_{rub}(v) = \frac{1}{\mathbf{degree}(v)} \sum_{\substack{(v,w) \in E \\ (w,v) \in E}} (X(w) - X(v))$$

If $F_{rub}(v) < 0$, we move the node v to the left by the amount $\min\{|F_{rub}(v)|, X(v) - X(u_l) - d(u_l, v)\}$, otherwise, we move the node to the right by the amount $\min\{|F_{rub}(v)|, X(u_r) - X(v) - d(v, u_r)\}$. Here u_l and u_r denote the left and right neighbor of v in its layer, and $d(u, v)$ is the minimal distance required between nodes u and v. It is easy to see that Z is decreased by these moves.

If the distance between two neighbored nodes of the same layer is minimal, we call the nodes *touching*. Touching nodes influence each other: if the left is drawn to the right and the right node is drawn to the left, none of both nodes can move. In order to get balance in this case, too, we use regions of nodes: touching nodes $v_1, \ldots v_n$ belong to the same region iff $P(v_1) < \cdots < P(v_n)$ and $F_{rub}(v_1) \geq \cdots \geq F_{rub}(v_n)$. The force at a region is

$$F_{rub}(\{v_1, \ldots, v_n\}) = \frac{1}{n} \sum_{i=1}^{n} F_{rub}(v_i)$$

We move all nodes of the region by $\min\{|F_{rub}(\{v_1, \ldots, v_n\})|, \text{available space}\}$. By these moves, Z decreases further.

```
(1)  while  Z is not satisfactory small do
(2)        for each layer V_i from  i = 1 to n do
(3)              Calculate all regions of V_i;
(4)              Move all nodes according to F_rub of their regions;
(5)        od
(6)  od
```

In the rubber band method, both predecessors and successors influence the position of a node at the same time. As a variation of this method, we can do downward and upward traversals of the layers where only the predecessor or only the successor positions are inspected. This model is more similar to a physical pendulum system. The nodes are like balls, the edges like strings. The uppermost balls are fixed at the ceiling. Then the pendulum system swings until the deflections are balanced. We define the predecessor force for downward traversals

$$F_{pendulate\_down}(v) = \frac{1}{\mathbf{indeg}(v)} \sum_{(w,v)\in E} (X(w) - X(v))$$

and the successor force for upward traversals

$$F_{pendulate\_up}(v) = \frac{1}{\mathbf{outdeg}(v)} \sum_{(v,w)\in E} (X(w) - X(v))$$

The construction of regions is the same as in the rubber band method. Although experiments show that this pendulum method decreases Z usually much faster than the rubber band method, Z does not decrease in each step. Thus, in practice, we combine both methods [Sa94].

[Sa96a] presents a variation of the pendulum method that enforces long edges (sequences of edges in the proper hierarchy) to be strictly vertical. Several other variants of layer-by-layer-sweep to position the nodes of a layer are described in [STT81, EaSu90] and [GKN93].

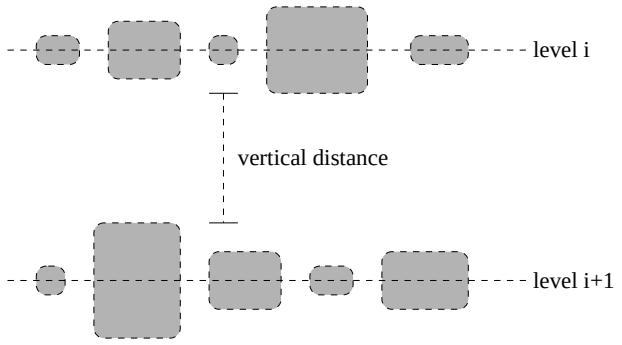
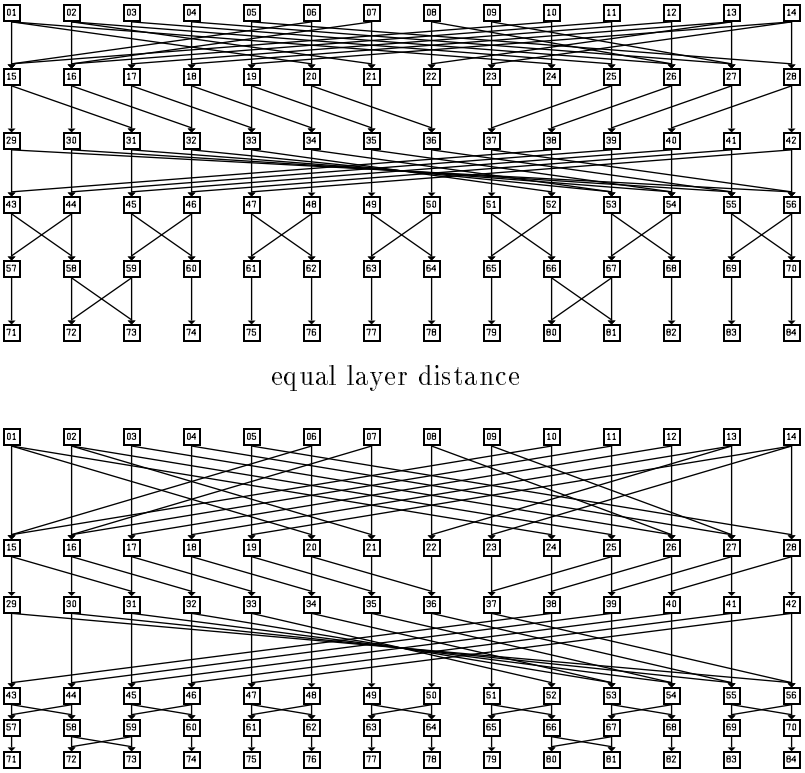


Figure 16: Vertical Positioning at the Levels

$Y(v)$ is calculated such that all nodes of the same layer are centered along a horizontal line (Fig. 16). There are two strategies to assign $Y(v)$:

- The vertical distance between layers is a constant δ: the layer V_i gets the reference line at $Y(V_i) = i\,\delta$.
- The vertical distance between two layers depends on the number of overlappings of the projection of the edges to the horizontal. Two different edges (v_1, w_1) and (v_2, w_2) overlap horizontally at one point between $X(v_1)$ and $X(w_1)$, iff $X(v_1) \leq X(v_2) \leq X(w_1)$ or $X(v_1) \leq X(w_2) \leq X(w_1)$. The maximal number of overlappings L_i between two layers V_i and V_{i+1} at any point can be calculated by a plane sweep in time $O(|V_i| + |V_{i+1}| + |E_i| + L_i)$ [Sa96b]. We calculate the reference lines top down: $Y(V_1) = \delta$ and $Y(V_i) = Y(V_{i-1}) + \delta\, L_{i-1}$.

The advantage of variable vertical distance between layers is that the angle of edges does not get too small. In particular, inhomogeneous dense graphs are more readable in this way (Fig. 17).



equal layer distance

variable layer distance

Figure 17: Layer Distance Strategies

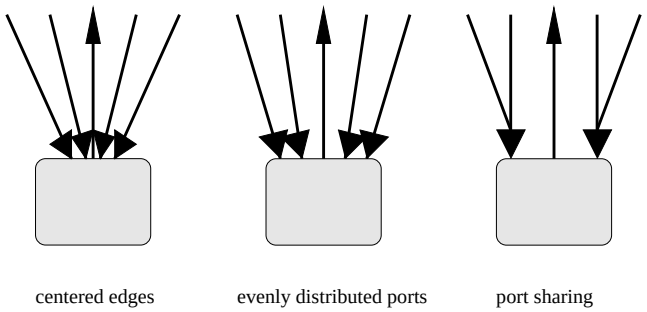


Figure 18: Edge Port Distribution

4.1.4 Phase 4: Positioning of edges

Start and end points of edges must be adjacent to the border of the corresponding nodes. These points at the border are called *edge ports*. There are several strategies to calculate edge ports:

- All edges point to the center of the node (Fig. 18, left). This is very easy to implement. Disadvantage: the ports may be so close together that arrowheads get lumpy and are not well readable.
- Each edge has its own edge port at the node (Fig. 18, middle). The ports are evenly distributed at the border. Such a distribution avoids concentrations of ports, if there are only few edges.
- Edges with the same orientation or style of arrowhead may share the same edge port (Fig. 18, right). The ports are evenly distributed at the border. This is even feasible if there are many edges, because edges share the arrowheads, too.

In the proper hierarchy, long edges are split into small edge segments and dummy nodes. This ensures that edges rarely cross nodes, because the dummy nodes don't overlap other nodes. Two situations may occur:

Due to the node positioning algorithm, the edge segments at a dummy node have (nearly) the same gradient. In this case, the dummy node can be removed and the edge can be replaced by a long segment that across several levels.

On the other hand, it may happen that a short edge segment still crosses a node. Then additional bend points are needed. This is the case if edges start at small nodes which are close to large nodes (Fig 19, left). It is obvious that for an edge (v, w) between adjacent layers, at most two additional bend points are needed (Fig 19, middle). As a variant, we can calculate for each angular edge two additional bend points such that the edge segments are oriented strictly horizontally or vertically. Then we get an orthogonal layout (Fig 19, right).

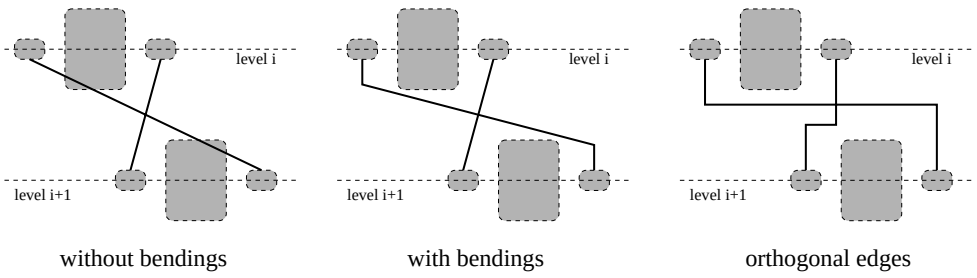


Figure 19: Bending of Edges

It is important that horizontal and vertical edges should not share segments, because otherwise the flow of the edges is not well visible. [Sa96a] presents a plane sweep method for the calculation of the additional bend points in time $O(|V_i| + |V_{i+1}| + |E_i| + k)$ where k is the number of rows of horizontal edge segments between layer i and layer $i + 1$.

The final result is a routing of edges such that edges never cross nodes. The drawing of an edge is a polygon. [Sa94] and [GKN93] present methods to convert this polygon into a sequence of splines with smooth transitions instead of bend points. Fig. 20 shows a PERT chart with spline edges.

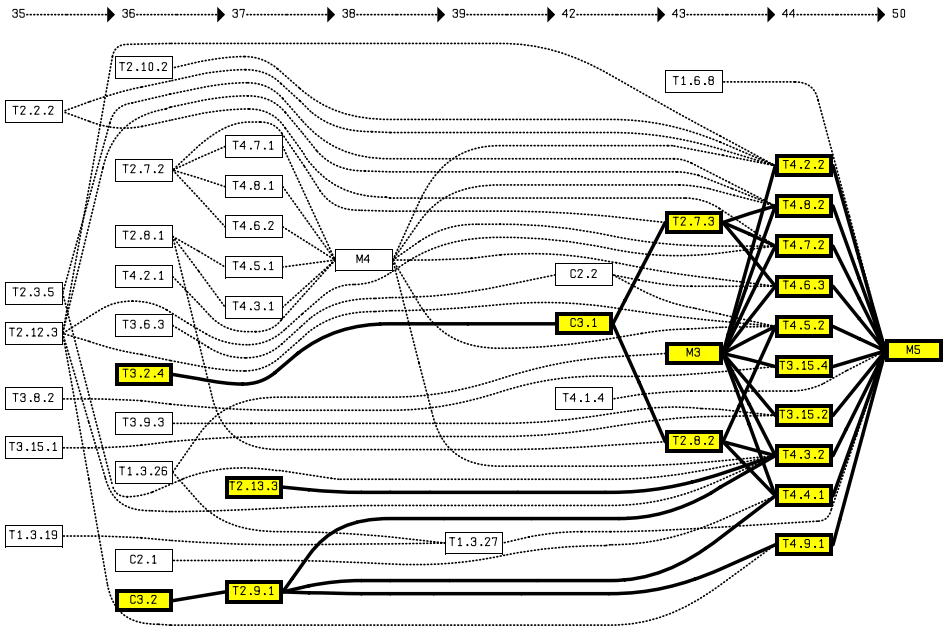


Figure 20: Spline Layout of PERT Chart

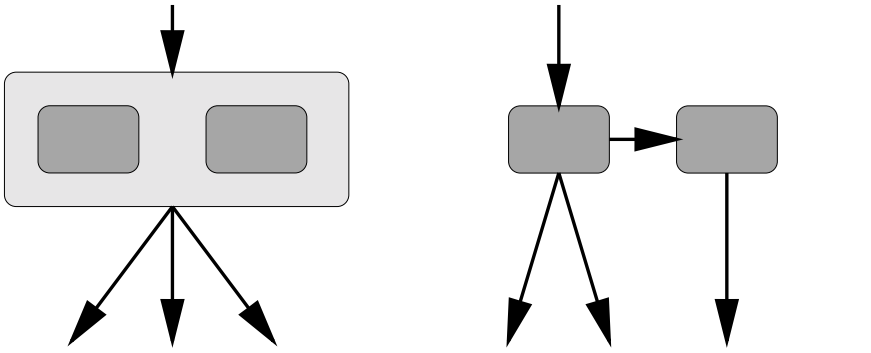
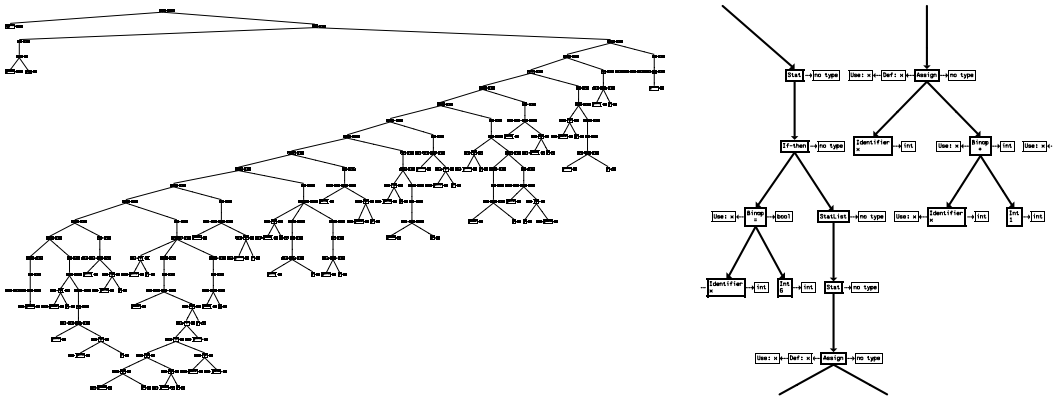


Figure 21: Neighbored Nodes

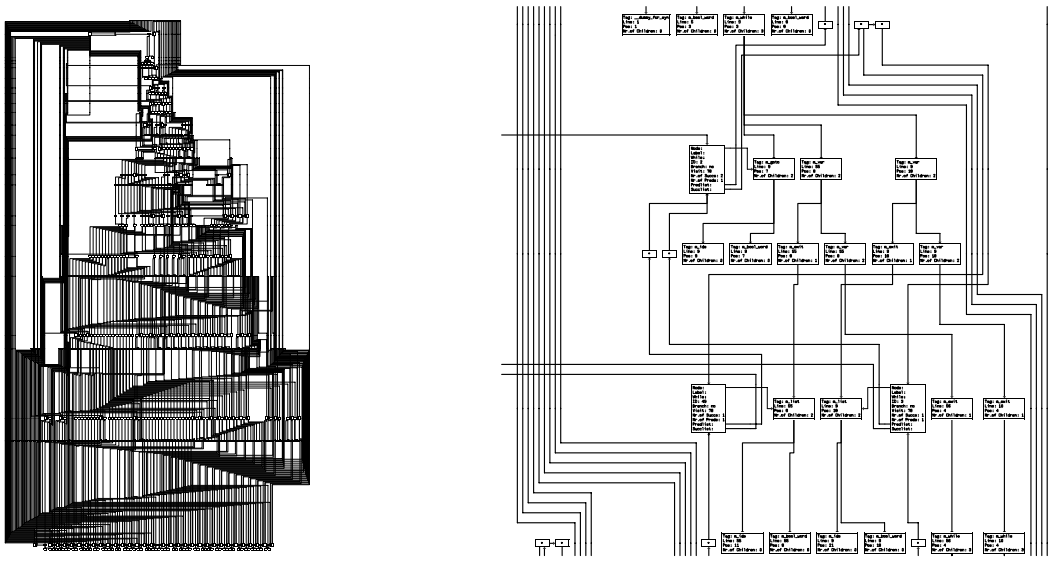
4.2 Application in Compiler Construction

The layer approach is mainly used to visualize the *directed* and the *dense* graphs that occur in compilers. The reason is the capability of the method to enforce uniform edge orientations and to avoid node overlappings. A compiler first parses the input program and checks the semantical rules of the programming language in a *frontend*. Usually, the intermediate program representation of the compiler frontend is a syntax tree annotated with attributes from the semantical analysis (e.g. types). Layout in layers produces good results for trees, where many simplifications of the algorithm can be done, e.g. partioning and crossing reduction for trees can be done simultaneously by only one depth first search traversal. The technical problem that annotations should occur as neighbors of the syntax nodes at the same level can be solved by combining neighbored nodes in phase 1 and 2 conceptually into one large node (Fig. 21). Fig. 22 shows a syntax tree annotated with two kinds of attributes: types and



Overview Detail

Figure 22: Annotated Syntax Tree



Overview Detail

Figure 23: Data Structure Graph (858 Nodes, 1109 Edges)

defined and used resources.

Typical compiler optimizations of the *middle end* use data flow analysis and work on procedure call graphs, annotated control flow graphs, or basic block graphs. The edges represent abstractions of the program flow. Together with various annotations such as data dependence edges, these graphs might become quite dense and complex. Control flow graphs are usually drawn with orthogonal edges (Fig. 14, left, Fig. 29). This convention comes from the flowchart diagram style of Nassi-Shneiderman.

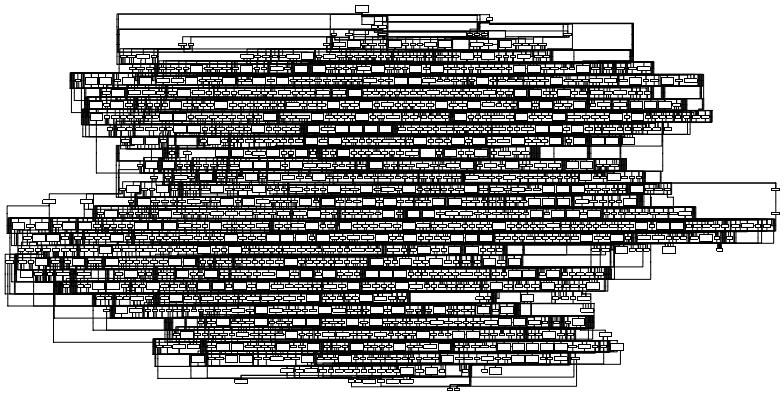
Data structure graphs show the details of the data structs used in the compiler. The nodes represent the structs containing several fields, and the edges visualize the pointers to the structs. Because pointers are related to certain fields, anchor point facilities are important, i.e. methods to specify the position of an edge port at a node. Because data structure graphs visualize many details, they are usually very large. Fig. 23 shows an example in overview and details.

4.3 Related Approaches

Woods presents an algorithm to draw planar graphs. This method has similiarities to layout in layers [Wo81]. Ranks $R(v)$ and relative positions $P(v)$ are calculated in one step such that the embedding has no edge crossings. This step is based on st-numbering, which is a very special way to number nodes of a graph. After this step, the normal positioning of nodes and edges can be applied as described in section 4.1.3 and 4.1.4. This way of rank calculation is applied preferably for undirected graphs, because it does not take edge orientation into account. The problem to find an embedding of a directed planar

graph where all edges point into the same direction is $\mathcal{NP}$-complete [GaTa94].

If the graph is not planar but not dense, planarization techniques can be used [BNT86, PST91]. In a first step, a large planar subgraph is calculated. The remaining edges are routed separately, such that only few edge crossings occur. There are efficient algorithms to calculate orthogonal layouts of fixed embedding settings of planar graphs on a grid [Ta87, TaTo89, FoKa96]. The main problem to find a maximal planar subgraph of a nonplanar graph, however, is $\mathcal{NP}$-hard [Jo82], such that heuristics must be used.



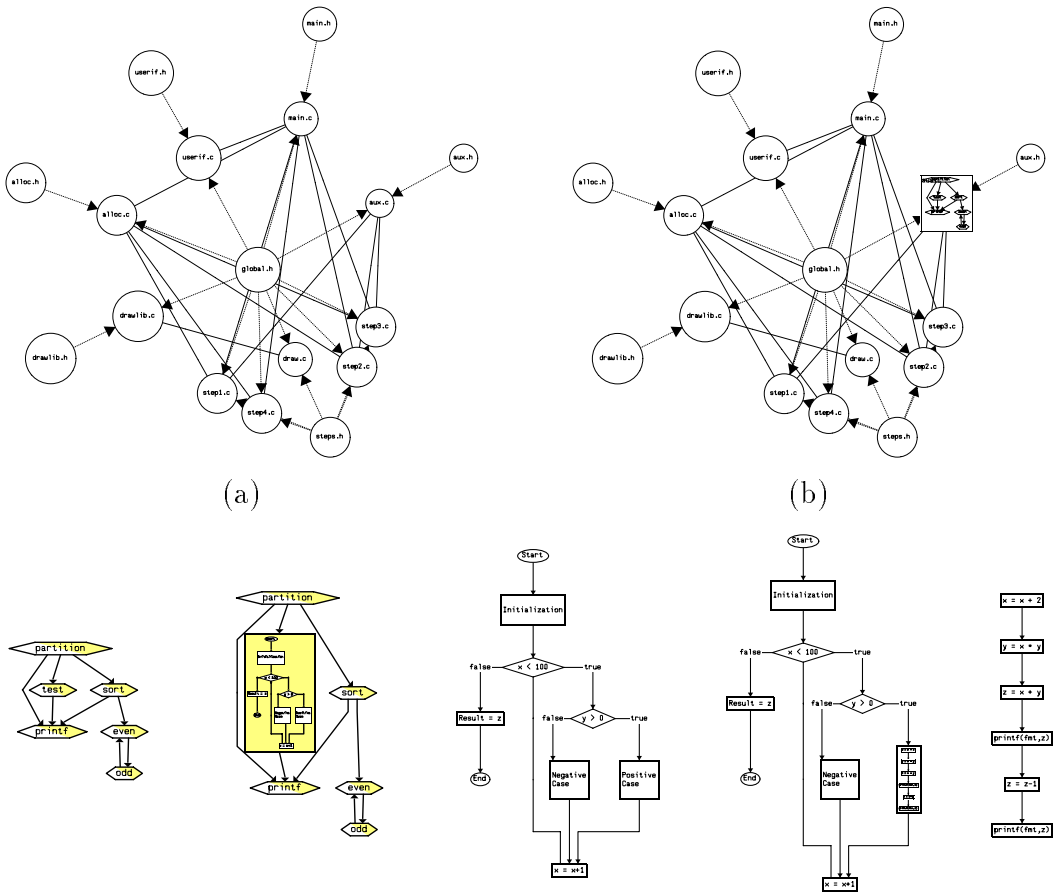
Example of a useless situation: flat graph with 1371 nodes, 3815 edges. The graph is rather dense, thus the layout is so narrow that only little is recognizable, unfortunately (Layout time: 18 sec. real time, Sparc 20).

Figure 24: Big, Flat Graph Without Structure

5 Grouping and Folding

Even if the layout algorithms are rather fast, there is a limit for the usability of flat graphs. If the size of a graph exceeds this limit, the layout algorithm takes a lot of time but the resulting picture of the graph is still unstructured with tangled edges (e.g. Fig. 24). Facilities are needed to stamp structures on the graph, to make them visible, to extract important parts or hide unimportant parts of the structures.

An example shows the main idea: A large program consists of many procedures with many statements. If we would visualize the control flow graph of all these statements at once, then we would see nothing but a black hole. But conceptually, the net of procedures is nested. All procedures are partitioned into the source files of the large program to be visualized. This fact can be exploited for visualization. At the first level, we show just the files as nodes (Fig. 25a). If a procedure of one file is used in another file, we draw an edge between those files. Multiple edges between the same nodes can be summarized to one thick edge, to improve the readability. To inspect the procedures



(a) file dependence graph: nodes represent the source files of the program (b) file dependence graph: one file is opened (c) procedure call graph of this file: nodes represent procedures (d) procedure call graph: one procedure is opened (e) basic block graph of this procedure: nodes are basic blocks (f) basic block graph: one block is opened (g) statement list of this block

Figure 25: Zooming into a Program Graph

of some file, we zoom into this file (Fig. 25b), i.e. we *unfold* the corresponding node. Then, we see the call graph of the procedures of this file. The nodes are the procedures and there is an edge from procedure A to B, iff A calls B (Fig. 25c). Next, we unfold one procedure (Fig. 25d) and see the basic block graph that shows the structure of the control flow of this procedure (Fig. 25e). To inspect statements of this graph, we select a basic block (Fig. 25f) and show its statement list *exclusively* (Fig. 25g). As we unfolded the graph, we can also fold the nodes in the inverse order.

It is also useful to see all statements at the same time. But then, it must be clearly which statement belongs to which procedure. We don't want to trust that the layout algorithm will place the nodes of the same procedure close

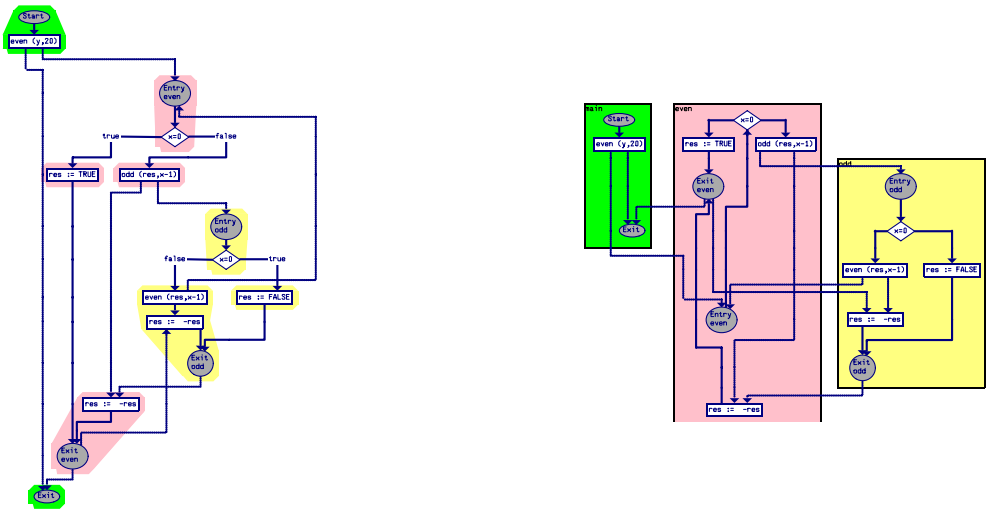


Figure 26: Interprocedural Control Flow Graph of 3 Procedures

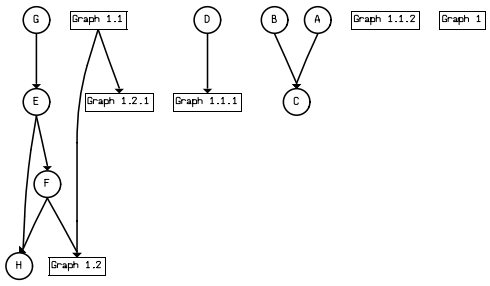
together by accident. A very simple method is to mark nodes by a unique colored wrapper (Fig. 26, left). Nodes that belong to the same procedure have the same color. Another possibility is to cluster the nodes, i.e. to calculate a layout such that the related nodes are so close together that a surrounding frame can be drawn (Fig. 26, right). In this case, the picture of a graph contains nested frames.

5.1 Compound Graphs and Dynamic Grouping

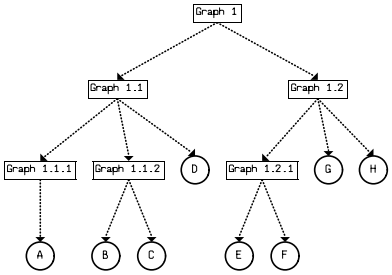
In all these cases, we don't deal any more with flat graphs $G = (V, E)$, but with *compound graphs*. A compound graph consists of a set V of primitive nodes, a set F of frames, a nesting relation $I \subseteq (V \cup F) \times (V \cup F)$ (inclusion relation) and a set of primitive edges $E \subseteq (V \cup F) \times (V \cup F)$. Since no frame can be nested into a primitive node or into itself, the nesting relation can be seen as a tree $T = (V \cup F, I)$ with $f \in F$ as inner nodes and $v \in V$ as leaves.

If the structure of the graph is static (as in applications such as Fig. 25 and 26) the nesting is defined in the graph specification. It is also useful to group nodes dynamically by user operations. For instance, during the analysis of large syntax trees, it is convenient to fold interactively parts of the tree that are currently not in the focus of interest (Fig. 28). Another example is to approximate paths of the control flow graph if only the reachability of statements but not the exact path between statements must be inspected (Fig. 29, middle and right). There are several possibilities for grouping selections:

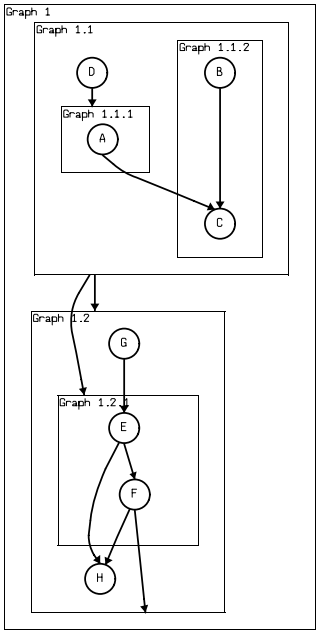
- Manual selection: point at individual nodes with the mouse, or drag a rectangle which contains all nodes to be selected, etc. If the group of



Edges E



Nesting Relation Tree T

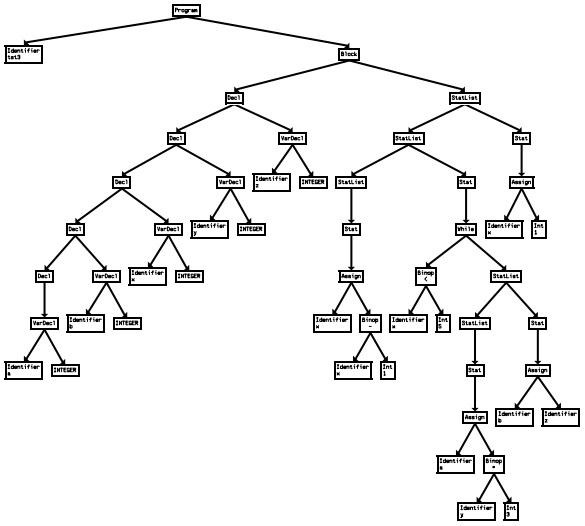


Picture of the Compound Graph

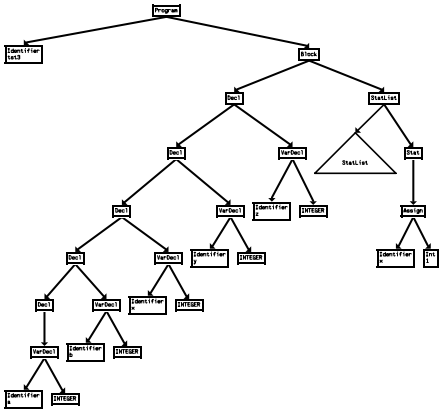
Figure 27: Compound Graph

nodes is very large and accidentally not placed closely together, manual selection is awkward and involved.

- Algorithmic selection: an algorithm to traverse the graph is used to collect the selected nodes. The user has only to select the kind of traversal.



Complete Syntax Tree



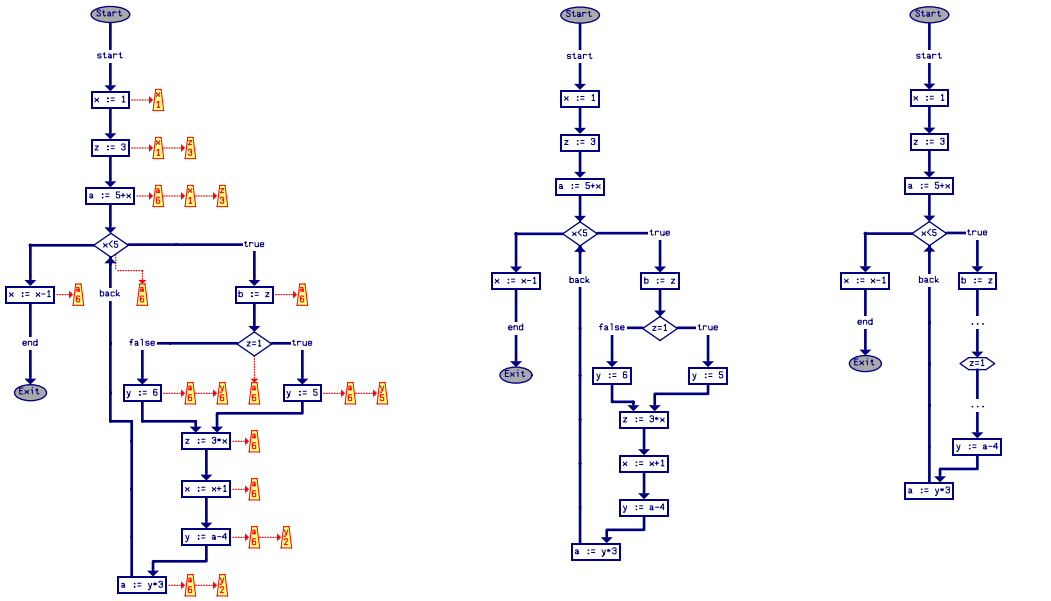
With Folded Subtree

Figure 28: Folding of Syntax Tree

Henry [He92] describes a system with a generic interface for selection of groups of nodes, and shows applications of algorithmic selections by reachability or shortest path algorithms. In compiler construction, the graphs are usually partitioned such that there are different classes of edges. For instance, the program graph of Fig. 25 is an interwoven compound graph consisting of edges of the classes file dependencies, procedure calls, and control flow. By including edge classes in the graph specification, it is possible to make detailed algorithmic selections. Examples:

- the *path region* of a set S of start nodes, a set T of end nodes and a class C is the set of nodes reachable from a start node $v \in S$ by a path of edges of class C which does not contain an end node $w \in T$. Folding parts of a control flow graph (Fig. 29, middle and right) is done by selecting the path region between two delimiting nodes, and collapsing it into one node.

- the *neighbor region* of S and C with radius n is the set of nodes reachable from a start node $v \in S$ by a path of edges of class C with the maximum length n. Folding a subtree (Fig. 28) is done by selecting the neighbor region of the subtree root node with radius ∞, and collapsing it into one node.



with Annotations Annotations hidden Compressed Path

Figure 29: Path Compression and Annotation Hiding in Control Flow Graph

Many compiler graphs have annotations, e.g. syntax trees with type attributes, control flow graphs with data flow information etc. In these cases, we have a main graph (tree, control flow graph) and smaller annotations (type trees, data flow lists) at each node of the main graph. To hide or expose all annotations at once, we select a node class. With hidden nodes, also all adjacent edges disappear (Fig. 29, left and middle).

5.2 Layout of Compound Graphs

There are several common layout methods for compound graphs. The recursive method is mostly used [PaTi90, No93, He92, MMPH96, Sa96b]:

(1) traversing the nesting tree T in postorder, **for each** $f \in F$ **do**
(2) layout graph consisting of the children of f in T
(3) compute bounding box of f
(4) **od**
(5) layout unnested nodes

The layout of each frame f is calculated independently. For the layout of the surrounding frame, f is considered as a large node. The advantages: (1) It is very simple to implement. (2) Each frame can use a specific layout algorithm. (3) If there is a change in frame f, it is not necessary to recalculate a complete layout. Only the frames on the path in T from the root to f are recalculated. The disadvantage: edges between nodes of different frames are not positioned properly, since the position of a node is calculated only with respect to the frame it belongs to.

The nondividing method [SuMi91, Sa96b] is more complex: It applies a layout method at once to all frames, and thus it is able to deal properly with edges crossing frames. It is a variant of the hierarchical layout algorithm by Sugiyama e.a. [STT81, EaSu90]:

1. Calculate a flat representation R of the compound graph. The flat representation is used to calculate the levels of the nodes such that most edges point downwards. It contains representatives of all nodes V and frames F. A frame $f \in R$ represents the upper border of the frame [SuMi91]; we can also add a second instance f' of f to R that represents the lower border [Sa96b]. A node v of a frame f must be positioned in between the borders f and f', which is represented by edges $f \rightarrow v \rightarrow f'$. A primitive edge e has an instance in R as it requires different levels of source and target nodes.

2. Calculate levels for the nodes and frame borders by sorting R topologically. If R is cyclic, some primitive edges are removed until R is acyclic. This is very similar to the partitioning phase of the normal hierarchical layout algorithm.

3. Normalize the representation. Edges crossing several levels are split into short edges and dummy nodes. For the dummy nodes, it must be decided which frame they belong to. Thus, [SuMi91] propose a proper compound digraph representation where nested frames are used instead of dummy nodes. [Sa96b] uses a simple heuristics by inspecting the frames of the start and end node of the edge.

4. At each level, permute the nodes in order to reduce edge crossings. This gives the relative position of the node. It is important that (a) all nodes belonging to a frame are in a consecutive sequence in the permutation, (b) the frames are not intertwined, i.e. the relative order of the frames is the same on all levels they occur. The crossing reduction is a recursive variant of the barycenter method.

5. Finally, calculate absolute positions of nodes and frames. Nodes of the same frame should be placed close together with a distance to the nodes of the other frames, such that a surrounding rectangle can be drawn. [Sa96b] uses a variant of the pendulum method in this step.

The advantage of this method: the layout shows the compound graph properly without overlappings. If there are edges from the outside of a frame to an inner node, then the placement of the node is not only influenced by the situation in the frame, but also by the global situation. The disadvantages: (1) It is relatively slow compared to the recursive divide-and-conquer method. (2) Every local change causes a global relayout. (3) Frames are not independent, thus all frames must be treated with the same layout parameters.

5.3 Graph Grammars

Grouping methods are closely related to graph grammars. While interactive grouping allows the selection of *arbitrary* sets of nodes, graph grammars are a mechanism for *rule based* selection of groups. Similar to context free string grammars, graph grammars consist of production rules that describe how a nonterminal node of a graph can be replaced. Fig. 30 shows an example grammar and a graph derivation. The application of a production rule is very similar to the unfolding of a collapsed graph.

It is possible to use the derivation of a graph to control the layout process. In this case, productions are annotated with layout rules. This is called a *layout graph grammar* [Br95]. For instance, in Fig. 30, there may be a layout rule that the subtree generated from terminal A must always be to the left of the subtree of B, while a general tree layout algorithm may permute the order of the subtrees in order to improve the balance of the tree. Layout graph grammars have been used in several systems [Hi95, BBH94, MSG95, ShMC96].

Since most compiler graphs are structured according to certain rules, layout graph grammars are quite appropriate. This gives syntax trees and control

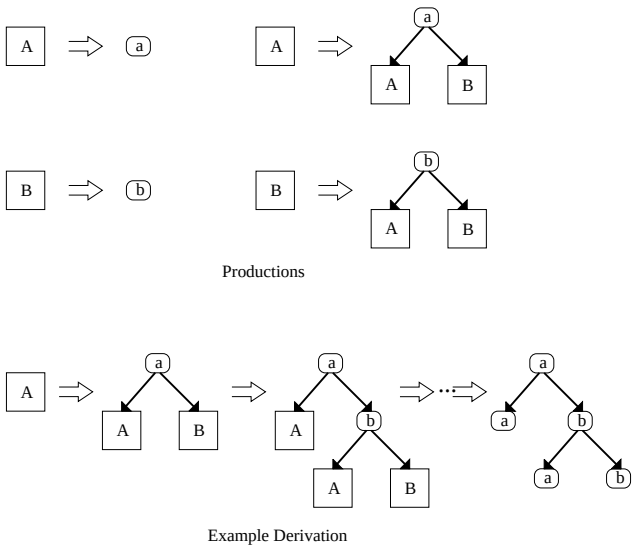


Figure 30: Graph Grammar of Binary Trees

flow graphs a uniform appearance that is easy to recognize. However, since the layout rules are local to a production, a layout method only based on graph grammars does not take the global structure of the graph into account. The results are rarely optimal wrt. used space, edge crossings, etc [Br95].

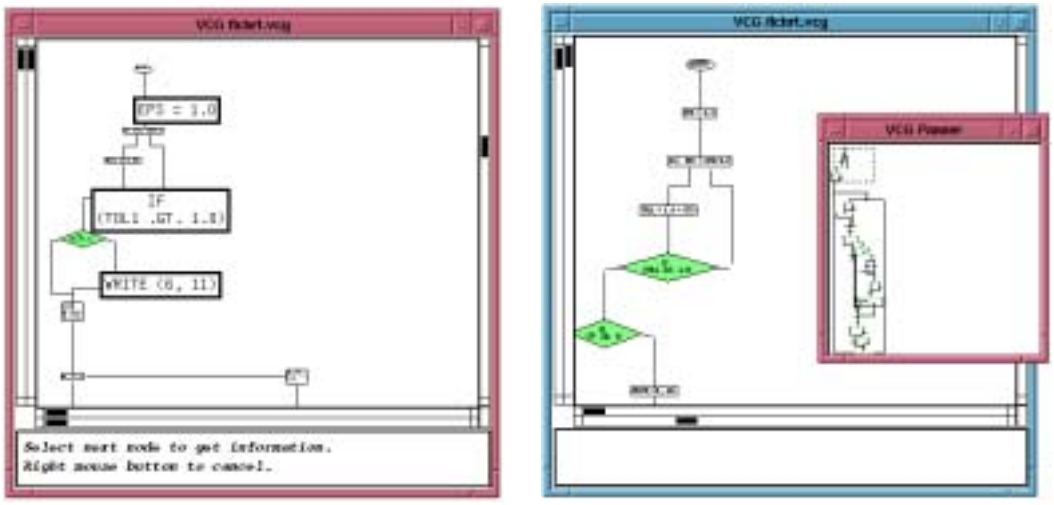
6 Browsing

A good graph layout tool does not only provide many fast layout algorithms, it also includes powerful interactive operations to browse the resulting picture. Usually, the layout is shown in a window on the screen. If the graph is very large and does not fit in the window, either only a part is visible or the picture must be shrunk. If the visible part is very small, the user often looses the orientation during the navigation through the graph. If the graph is scaled too much, details, e.g. labels of nodes, are not readable anymore.

6.1 Linear Views

In a linear view, the picture is uniformly scaled. The relation between picture and original layout is linear. There are several possibilities to solve the conflict between detailed and full view:

- *Overview while details can be selected:* The main window shows the shrunk layout. Labels of nodes or edges can be made visible by selecting them. Then, boxes appear with the labels in normal size. However, these boxes overlap and hide parts of the picture (Fig. 31, left).



Selection of Details Multiple Windows for Overview

Figure 31: Browsing Methods

- *Detailed view with panner:* The main window shows a part of the layout in normal magnification. A second window *(panner)* shows an overview. Positioning of the visible part of the main window can be done by selecting rectangles in the overview window (Fig. 31, right).

6.2 Fisheye Views

Fisheye views show the point of interest in detail and the overview of the graph in the same window. This is done by distorting the picture. The picture is scaled nonuniformly. Objects far away from the focus point are shrunk while objects near the focus point are magnified. The degree of visual distortion depends on the distance from the focus point. The visual effect is very similar to the fisheye lenses in photography (Fig. 32, right).

Fisheye views were inspected by [Fu86, SaBr94, No93, KRB95, MiSu91, FoKe96, StMu96, CCFS96]. They can be divided into *graphical fisheye views*, where the distance from the focus point is a function of the coordinates (e.g. the Euclidean distance), and *logical fisheye views*, where the distance is any logical function wrt. the graph (e.g. the length of the shortest path between focus point and node). A fisheye view might be *distorting*, i.e. objects far away from the focus are shrunk, and *filtering*, i.e. unimportant objects far away from the focus point are hidden. Further, a fisheye view is *layout independent* [No93], if first the demagnification or filtering is calculated and then the layout is done. Otherwise, it is *layout dependent*. Layout independent fisheye views have the advantage that the layout can be calculated using the knowledge which nodes are shrunk or filtered. This resembles the folding mechanism in that it saves space in the layout. Graphical fisheye views must be layout dependent, because in order to calculate the distance by coordinates, the layout must be known.

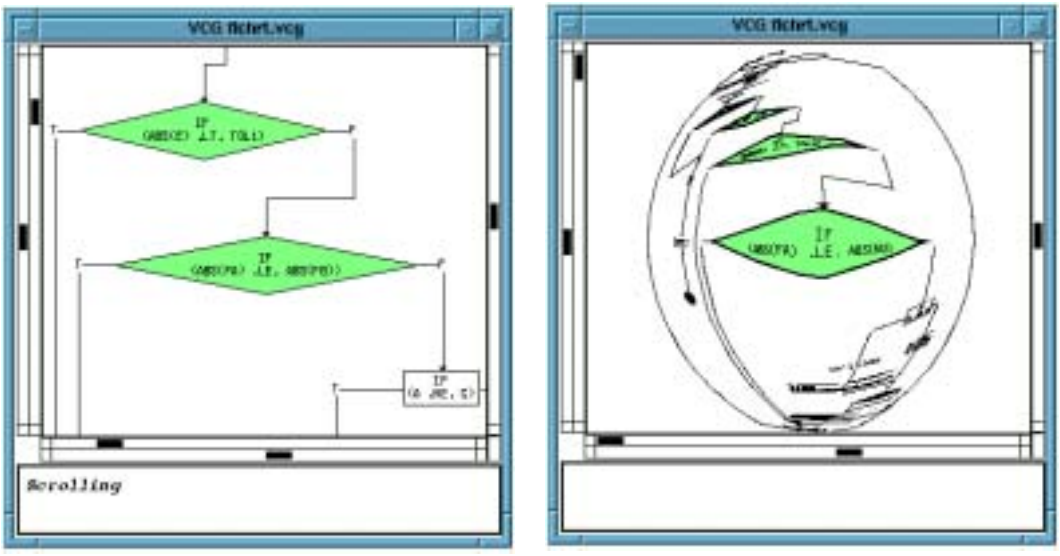
6.2.1 Distorting Fisheye Views

Graphical fisheye views are based on a bijective transformation function h that describes the mapping of the distances from the focus f_l in the layout into distances from the focus f_p in the picture. General rules are:

- $h(0) = 0$: The focus point in the layout is mapped to the focus point of the picture.
- h must be strictly increasing: Points cannot overtake during the transformation, i.e. points in the layout being closer to f_l must be mapped to points being closer to f_p as well.
- h must be bijective. Fisheye views must not only be drawn, but also react on mouse picks. Thus the inverse function must exist.

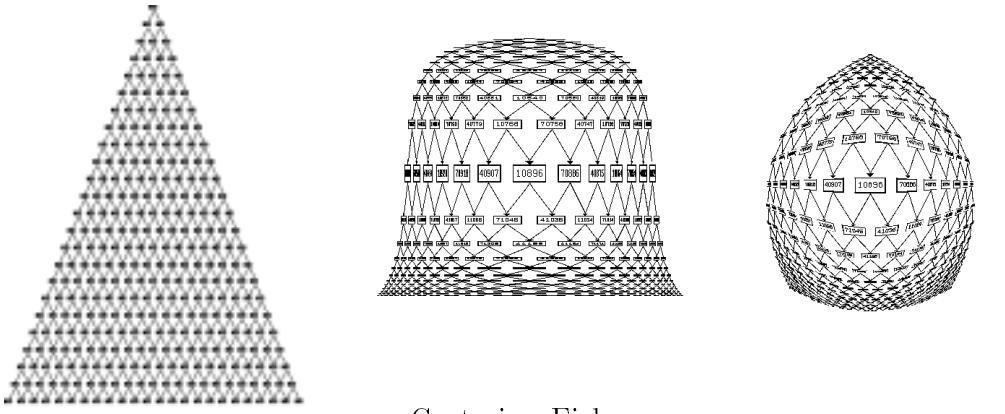
If $h(x) > x$ for all points $x > 0$, then the focus point is magnified, if $h(x) < x$ the focus point is demagnified. The magnification at distance x from the focus point is just $\frac{\partial h}{\partial x}(x)$. Transformation functions commonly used for fisheye views are

$$\begin{aligned} h(x) &= \frac{Kx}{Ax+1} \\ h(x) &= K\,\sin(Ax) \quad \text{with } x \in [0, \tfrac{\pi}{2A}] \\ h(x) &= K\,\arctan(Ax) \end{aligned}$$



Linear View Fisheye View

Figure 32: Different Views



Linear View | Cartesian Fisheye View | Polar Fisheye View

Figure 33: Graph with Different Views

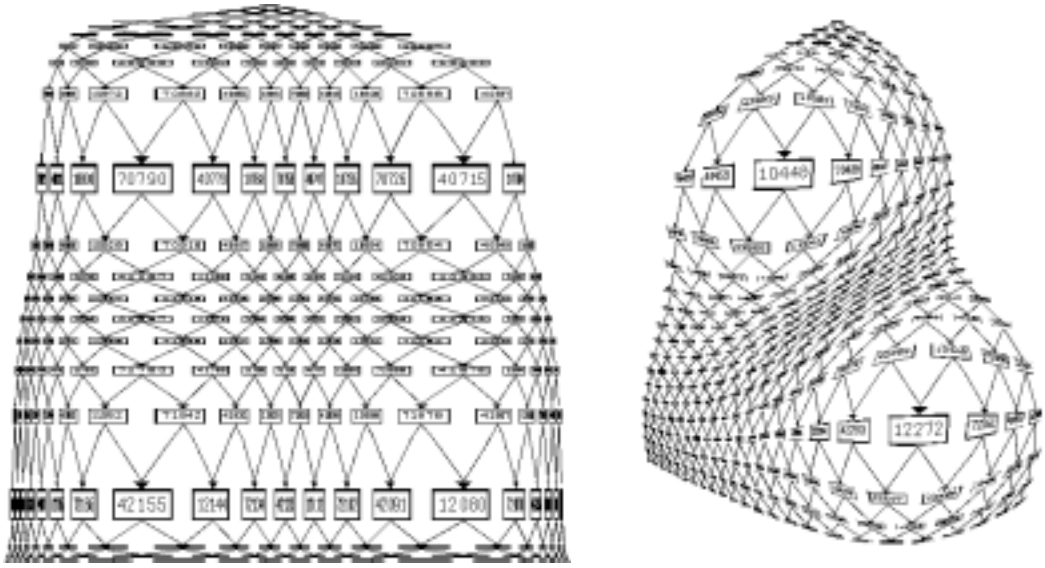
K and A allow to select the magnification at focus point, and the radius of interest. The Cartesian fisheye view applies h independently to the x and y directions: $(x, y) \rightarrow (h(x), h(y))$. Polar fisheye views are based on the polar coordinates. h is applied to the distance, and the angle of the ray though the origin remains: $(d, \phi) \rightarrow (h(d), \phi)$. Cartesian views are invariant with respect to horizontal and vertical lines, thus they are appropriate for orthogonal drawing. Polar views however are more closer to the fisheye lenses of the photography.

The idea of a fisheye is to make the area near the focus point well visible. A distortion near the focus point is often unwelcome. Thus, it is better to use *focus areas* instead of *focus points*. Inside the focus area, there is a linear magnification without distortion. The simplest way is to define a transformation in two parts, e.g.:

$$h(x) = \begin{cases} Kx & \text{for} \quad x \leq a_l \quad \text{Here is a linear scaling.} \\ \frac{K(x-a_l)}{A(x-a_l)+1} + a_p & \text{for} \quad x > a_l \quad \text{Here is a distortion.} \end{cases}$$

a_l is the radius of the focus area in the layout, and $a_p = Ka_l$ is the radius of the focus area in the picture. With this simple method, we get a focus square for Cartesian fisheye views and a focus circle for polar fisheye views. Recently, fisheye views with arbitrary focus polygons were developed [FoKe96]. These methods are more complex and require the calculation of Voronoi diagrams.

Another extension is the usage of multiple fisheye points [KRB95, MiSu91]. These are implemented by converting each display point of the graph once for every focus point and then taking the mathematical average of the transformed points as the picture location. Due to the special construction, superposition of two Cartesian views introduces two focus points but also two mirror focus points (Fig. 34). The mirror focus points are located an the further corners of the rectangle whose diagonal is given by the normal focus points. So there



Cartesian fisheye view

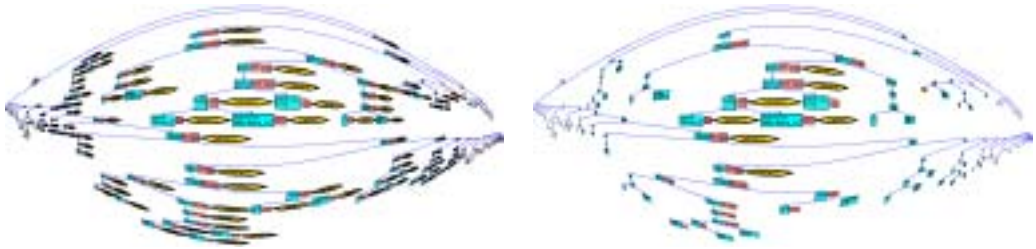
Polar fisheye view

Figure 34: Fisheye Views with Two Focus Points

are four points where the magnification is maximal. This effect does not occur with polar fisheyes.

6.2.2 Filtering Fisheye Views

Filtering fisheyes [SaBr94, Sa96b] show many details at the focus point, but they filter graphical objects that are far from the focus point. This improves the visibility of the main structure, which would probably go lumpy with all the shrunk, unimportant details far from the focus point. Thus, objects are filtered according to their *visual worth*. The visual worth depends on the distance to the focus point and on an *a priori importance* (api) of the nodes and edges of a graph, which is given in the graph specification. For instance, in an attributed syntax tree, the main structure is the tree, thus it has an larger api than the attributes. The user can select the threshold *level of detail* (lod) to influence



no filtering, lod = 0

with filtering, lod = 130

Figure 35: Polar Filtering Fisheye View of Attributed Syntax Tree

the amount of visible objects. An object is visible if $h(x, \text{api}) > \text{lod}$, where x is the distance to the focus point and h is the function calculating the visual worth. Properties of h:

- $h(x, \text{api}) \leq h(x', \text{api})$ if $x \geq x'$. The function h is monotonic decreasing wrt. distances. Objects far from the focus point are less interesting, since the focus point is the point of interest.
- $h(x, \text{api}) \leq h(x, \text{api}')$ if $\text{api} \leq \text{api}'$. The function h is monotonic increasing wrt. api. Objects with small api are less important and can be preferred for filtering.

A function commonly used for filtering fisheyes (S(x) is the transformed size of a node in distance x in the picture, the parameters $c, d, e > 0$):

$$h(x, \text{api}) \quad = \quad c\, S(x) \text{api}^d + e$$

6.2.3 Logical Fisheye Views

On logical fisheye views, the distance is not calculated wrt. coordinates but with respect to the structure of the graph. Distorting and filtering views are possible. The typical distance is the length of the shortest path from the focus node [Fu86]. For compound subgraphs, a combined method must be used taking into account the primitive edges and the nesting structure [No93]. The reason: a node should not be larger (or filtered later) than the frame it belongs to. Logical fisheye views have two advantages:

- They reflect the structure of the graph, because a logical fisheye view does not depend on the node positions. A graphical fisheye might filter a node that is closely related to the focus node by the fact that it is accidentally placed far away from the focus point.
- They allow to calculate the layout after the fisheye effect. Layout calculation becomes the faster the more nodes are filtered away. Furthermore, the space occupation might be better if the layout is calculated afterwards.

As disadvantage, logical fisheye views don't have similarities with optical physics. Human beings are not used to deal with such effects. For instance, moving the focus point of a logical fisheye view might change the graph so much that the layout afterwards cannot be compared with the layout before.

6.2.4 3-D approaches

We described fisheye views as two dimensional transformation. However, the fisheye picture of a graph especially with graphical polar view looks like a projection of the 2 D drawing space into 3 D (e.g., a sphere). The focus point seems to be near to the viewer of the picture, thus it is enlarged. There are true three dimensional approaches [CCFS96, MRC91]: Instead of a distortion function, a mapping into 3 D (e.g., onto a surface) is provided with a viewpoint of a synthetic camera. The use of an underlying grid and shading technics improves the 3 D effect. On the other hand, the exploration of the graph might be slightly more complex since the user has to navigate through 3 D and control the surface at the same time.

7 Conclusion

We have described methods for interactive graph visualization in the application domain *compiler construction*. Most heuristics which we presented are implemented in the VCG tool [Sa94] and are successfully used as debugging aid in a commercial compiler project [AAS94] and in teaching at the university. Since the VCG tool is publicly available, we know also about applications of the tool ranging from the generation of genealogical trees up to circuit design and debugging tools. The tool seems to fit to many more application areas. Some similar visualization tools exist [Hi95, FrWe93, GKN93, Sc95] that focus on different areas.

How useful is a visualization tool, in compiler construction or in general? We believe that the success of such a tool does not only depend on the quality of the graph layout algorithm, but also very much on the facilities of the user interface. Powerful browsing methods simplify the interactive graph exploration and are absolutely necessary for the acceptance of visualization. The implementation of a comfortable user interface means a considerable amount of work, and unfortunately, this is often neglected. Another important factor for the usability of an interactive tool is its speed. This, however, is a never ending story: as visualization tools become faster the graphs get larger that are dealt with.

There are many empirical studies about the usefulness of program visualization (for an overview, see [Hy93]). These take into account psychological effects, such as time pressure during debugging, education and familiarity of the subjects of the tests with visualization techniques. The results vary a lot. Although most experiments found graphical representations better, others made just the contradictory observation [GPB91]. The usability of graphical representations of data and programs can not be assured in the general case. It depends on the knowledge and expectations of the users (in many experiments, the subjects are students), on the aim of the visualization, on the visualiza-

tion method (static visualization or animation), and on the capabilities of the visualization tool.

We think that in the research community of compiler construction, visualization of compiler data structures is widely accepted. This may be influenced by the fact that advanced compiler construction is usually taught by using graph theoretical terminology: data structures in compilers are graphs. Thus, the compiler construction community is familiar with graphs. Our experience is that visualization allows better understanding of the behavior of compilers, if suitable layout strategies and powerful browsing methods are used.

8 Acknowledgments

We like to thank P. Bouillon, R. Heckmann and R. Wilhelm for their comments on the presentation of this survey.

References

[AAS94] Alt, M.; Aßmann, U.; Someren, H.: Compiler Phase Embedding with the CoSy Compiler Model, *in* Fritzson, P.A., ed.: Compiler Construction, Proc. 5th International Conference CC'94, Lecture Notes in Computer Science 786, pp. 278-293, Springer, 1994

[AlGo89] Almasi, G.S.; Gottlieb, A.: Highly Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1989

[BBH94] Bachl, W.; Brandenburg F.J.; Hickl T.: Hierarchical Graph Design Using HiGraD, Technical Report MIP 9405, Fakultät Mathematik und Informatik, University of Passau, Germany, 1994.

[BHR96] Brandenburg, F.J.; Himsolt, M.; Rohrer, C.: An Experimental Comparison of Force-Directed and Randomized Graph Drawing Algorithms, *in* [Br96], pp. 76-87, 1996

[BNT86] Batini, C.; Nardelli, E.; Tamassia, R.: A Layout Algorithm for Data Flow Diagrams, IEEE Trans. on Software Engineering, SE-12(4), pp. 538-546, 1986,

[BoPr91] Boehm, W.; Prautzsch, H.: Numerical Methods, A.K. Peters, Vieweg, 1991.

[Br93] Bräunl, T.; Parallele Programmierung, Eine Einführung, Vieweg, 1993.

[Br95] Brandenburg, F.J.: Designing Graph Drawings by Layout Graph Grammars, *in* [TaTo95], pp. 416-427, 1995.

[Br96] Brandenburg, F.J., ed.: Proc. Symposium on Graph Drawing, GD'95, Lecture Notes in Computer Science 1027, Springer, 1996

[Ca80] Carpano, M.J.: Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis, IEEE Trans. Sys., Man, and Cybernetics, SMC 10(11), pp. 705-715, 1980.

[CCFS96] Carpendale, M.S.T.; Cowperthwaite, D.J.; Fracchia, F.D.; Shermer T.: Graph Folding: Extending Detail and Context Viewing into a Tool for Subgraph Comparisons, *in* [Br96], pp. 127-139, 1996

[DaHa89] Davidson, R.; Harel, D.: Drawing Graphs Nicely Using Simulated Annealing, Technical Report CS89-13, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1989

[Ea84] Eades, P.: A Heuristic for Graph Drawing, Congressus Numerantium 41, pp. 149-160, 1984.

[EaKe86] Eades, P.; Kelly, D.: Heuristics for Reducing Crossings in 2-Layered Networks, Ars Combinatorica 21-A, pp. 89-98, 1986.

[EaSu90] Eades, P.; Sugiyama, K.: How to Draw a Directed Graph, Journal of Information Processing, 13 (4), pp. 424-437, 1990

[EaWo86] Eades, P.; Wormald N.: The Median Heuristic for Drawing 2-Layers Networks, Technical Report 69, Department of Computer Science, University of Queensland, 1986

[FLM95] Frick, A.; Ludwig, A.; Mehldau, H.: A Fast Adaptive Layout Algorithm for Undirected Graphs, *in* [TaTo95], pp. 388-403, 1995

[FoKe96] Formella, A.; Keller, J.: Generalized Fisheye Views of Graphs, *in* [Br96], pp. 242-253, 1996

[FoKa96] Fößmeier, U.; Kaufmann, M.: Drawing High Degree Graphs with Low Bend Numbers, *in* [Br96], pp. 254-266, 1996

[FrRe91] Fruchterman, T.M.J.; Reingold, E.M.: Graph Drawing by Force-Directed Placement, Software – Practice and Experience 21, pp. 1129-1164, 1991

[FrWe93] Fröhlich, M,; Werner, M.: Das interaktive Graph Visualisierungssytem daVinci V1.2, Technical Report *(in German)*, Fachbereich Mathematik und Informatik, University of Bremen, Germany, 1993

[Fu86] Furnas, G.W.: Generalized Fisheye Views, Proc. ACM SIGCHI'86, Conference on Human Factors in Computing Systems, pp. 16-23, 1986

[GaJo79] Garey, M.R.; Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman & Co., San Francisco, 1979

[GaJo83] Garey, M.R.; Johnson, D.S.: Crossing Number is NP-complete, SIAM Journal of Algebraic and Discrete Methods, 4(3), pp. 312-316 1983

[GaTa94] Garg, A.; Tamassia, R.: On the Computational Complexity of Upward and Rectilinear Planarity Testing, Technical Report CS-94-10, Department of Computer Science, Brown University, 1994

[GKN93] Gansner, E.R.; Koutsofios, E.; North, S.C.; Vo, K.-P.: A Technique for Drawing Directed Graphs, IEEE Trans. on Software Engineering, 19(3), pp. 214-230, 1993

[GPB91] Green, T.R.; Petre, M.; Bellamy, R.K.E.: Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the Match-Mismatch Conjecture, Fourth Workshop on Empirical Studies of Programmers, pp. 121-146, 1991

[HaSa93] Harel, D.; Sardas, M.: Randomized Graph Drawing with Heavy-Duty Preprocessing, Technical Report CS93-16, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 1993

[He92] Henry, T.R.: Interactive Graph Layout: The Exploration of Large Graphs, Ph. D. Thesis, TR 92-03, Department of Computer Science, The University of Arizona, 1992

[Hi95] Himsolt, M.: GraphEd – A Graphical Platform for the Implementation of Graph Algorithms, *in* [TaTo95], pp. 182-193, 1995.

[Hy93] Hyrskykari, A.: Development of Program Visualization Systems, Report, Department of Computer Science, University of Tampere, Finland, *presented at the 2nd Czech British Symposium of Visual Aspects of Man-Machine Systems, Praha,* 1993

[Jo82] Johnson, D.: The NP-completeness column: An ongoing guide, Journal on Algorithms, 3(1), pp. 215-218, 1982

[JuMu96] Jünger, M.; Mutzel, P.: Exact and Heuristic Algorithms for 2-Layer Straightline Crossing Minimization, *in* [Br96], pp. 337-348, 1996

[KaKa89] Kamada, T.; Kawai, S.: An Algorithm for Drawing General Undirected Graphs, Information Processing Letters, 31, pp. 7-15, 1989.

[KRB95] Kaugars, K.; Reinfels, J.; Brazma A.: A Simple Algorithm for Drawing Large Graphs on Small Screens, *in* [TaTo95], pp. 278-281, 1995

[Ma92] Masui, T.: Graphic Object Layout with Interactive Genetic Algorithms, Proc. IEEE Workshop on Visual Languages, pp. 74-80, 1992

[Me84] Mehlhorn, K.: Data Structures and Algorithms, Vol. 2: Graph Algorithms and NP-Completeness, Springer, 1984,

[MiSu91] Misue, K.; Sugiyama K.: Multi-viewpoint Perspective Display Methods: Formulation and Application to Compound Graphs, Human Aspects in Computing, *in* Bullinger H.J., ed.: Proc. 4th Intern. Conf. on Human-Computer Interaction pp. 834-838, Elsevier, 1991

[MMPH96] Madden, B.; Madden, P.; Powers, S.; Himsolt, M.: Portable Graph Layout and Editing, *in* [Br96], pp. 385-395, 1996

[MRC91] Mackinlay, J.D.; Robertson, G.G.; Card, A.K.: The Perspective Wall: Detail and Context Smoothly Integrated, *in* Proc. ACM SIGCHI'91, Conference on Human Factors in Computing Systems, pp. 173-179, 1991

[MSG95] McCreary C.L.; Shieh F.S.; Gill H.: CG: A Graph Drawing System Using Graph Grammar Parsing, *in* [TaTo95], pp. 270-273, 1995.

[No93] Noik, E. G.: Layout-independent Fisheye Views of Nested Graphs, Proc. IEEE Symposium on Visual Languages, pp. 336-341, 1993

[Pa89] Pazel, D.: A Graphical Interface for Evaluating a Genetic Algorithm for Graph Layout, Technical Report RC 14348, IBM T.J. Watson Research Center, 1989

[PaTi90] Paulisch, F. N.; Tichy, W.F.: EDGE: An Extendible Graph Editor, Software – Practice and Experience 20 (S1), pp. 63-88, 1990

[PST91] Protsko, L.B.; Sorenson, P.G.; Tremblay, J.P.; Schaefer, D.A.: Towards the Automatic Generation of Software Diagrams, IEEE Trans. on Software Engeneering, 17(1), pp. 10-21, 1991,

[QuBr79] Quinn Jr., N. R.; Breuer, M. A.: A Force Directed Component Placement Procedure for Printed Circuit Boards, IEEE Trans. on Circuits and Systems, CAS-26(6), pp. 377-388, 1979.

[Sa94] Sander, G.: Graph Layout through the VCG Tool, Technical Report A03-94, FB 14 Informatik, University of Saarbrücken, Germany, 1994, *an extended abstract is in* [TaTo95], pp. 194-205, 1995.

[Sa96a] Sander, G.: A Fast Heuristic for Hierarchical Manhattan Layout, *in* [Br96], pp. 447-458, 1996

[Sa96b] Sander, G.: Visualisierungstechniken für den Compilerbau, Doctoral Thesis, *to appear in German*, FB 14 Informatik, University of Saarbrücken, Germany, 1996

[SaBr94] Sarkar, M.; Brown, M. H.: Graphical Fisheye Views, Communications of the ACM, vol. 37, no. 12, pp. 73-84, 1994

[Sc95] Scott, A.: A Survey of Graph Drawing Systems, Technical Report 95-1 Department of Computer Science, University of Newcastle, Australia, 1995

[ShMC96] Shieh F.-S.; McCreary C.L.: Directed Graphs Drawing by Clan-based Decomposition, *in* [Br96], pp. 472-482, 1996

[StMu96] Storey, M.D.; Müller, H.A.: Graph Layout Adjustment Strategies, *in* [Br96], pp. 487-499, 1996

[STT81] Sugiyama, K.; Tagawa, S.; Toda, M.: Methods for Visual Understanding of Hierarchical Systems, IEEE Trans. Sys., Man, and Cybernetics, SMC 11(2), pp. 109-125, 1981.

[SuMi91] Sugiyama K.; Misue K.: Visualization of Structural Information: Automatic Drawing of Compound Digraphs, IEEE Trans. Sys., Man, and Cybernetics, 21(4), pp. 876-892, 1991.

[SuMi94] Sugiyama, K.; Misue, K.: Graph Drawing by Magnetic-Spring Model, Research Report ISIS-RR-94-14E, Inst. Social Information Science, Fujitsu Labs. Ltd., 1994

[SuMi95] Sugiyama, K.; Misue, K.: A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm, *in* [TaTo95], pp. 364-375, 1995

[Ta87] Tamassia, R.: On Embedding a Graph in the Grid with the Minimum Number of Bends, SIAM Journal of Computing, 16(3), pp. 421-444, 1987.

[TaTo89] Tamassia, R., Tollis, I.G.: Planar Grid Embedding in Linear Time, IEEE Trans. on Circuits and Systems, 36(9), pp. 1230-1234, 1989.

[TaTo95] Tamassia, R.; Tollis, I.G., eds.: Graph Drawing, Proc. DIMACS Intern. Workshop GD'94, Lecture Notes in Computer Science 894, Springer, 1995.

[Tu94] Tunkelang, D.: A Practical Approach to Drawing Undirected Graphs, Technical Report CMU-CS-94-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1994

[Wa77] Warfield, J. N.: Crossing Theory and Hierarchy Mapping, IEEE Trans. Sys., Man, and Cybernetics, SMC 7(7), pp. 505-523, 1977.

[WiMa95] Wilhelm, R.; Maurer, D.: Compiler Design, Addison Wesley, 1995

[Wo81] Woods, D.R.: Drawing Planar Graphs, Technical Report STAN-CS-82-943, Computer Science Department, Stanford University, 1982