ALGORITHM FOR GRAPHIC LAYOUT IN VIFOR

Vaclav Rajlich
Nicolas Damaskinos

Department of Computer Science
Wayne State University
Detroit, MI 48202

VIFOR is a tool for maintenance of large FORTRAN programs. It contains a database which stores information on all nonlocal declarations of the programs (i.e. subroutines, functions, commons), all source files, and all relations among them, see [1] for a detailed description.

The programmer accesses this database by queries which produce views. Each view is a subset of the information stored in the database. VIFOR displays these views in browsers, which are specialized windows displaying the views graphically.

Graphical display of the program structure has been a research problem of increasing interest. Such graphs are usually complex and their readability is a constant problem [2]. The distinguishing feature of VIFOR layouts is the placement of all nodes into two columns depending on their class. We shall refer to such a layout as Two Column Graph (2CG). Nodes representing processes (subroutines, functions and main program) are displayed in the left column and nodes representing global data (commons) are displayed in the right column. The call relations among the processes are drawn as bent edges and appear to the left of the process column. The horizontal part of a call-edge emanates from the calling process and the vertical part extends to the position of the farthest node being called, with arrows pointing to the called nodes. Reference relations, i.e. relations between commons and processes that use them, are drawn as straight lines between pairs of nodes in the respective columns. Each 2CG is displayed in a special window called Browser (Figure 1).

The relative simplicity of the data model in VIFOR enables us to use the 2CG. The 2CG layouts strike a balance between efficiency and readability. They are efficient in terms of execution speed because they are essentially one-dimensional, with the length of columns being the main parameter. Nodes and relations are drawn in distinct rectangular regions of the drawing plane, without interference between them. The placement of nodes determines the routing of all relations in the layout.

The 2CG consists of two subgraphs, the call graph and the reference graph. Our algorithm first builds the call graph by determining the positions of the nodes in the process column, and then appends the reference graph to the drawing. In the next section, we describe the part of the algorithm that builds the layout of the call graph. The algorithm tries to communicate as much information about the call graph as possible. The space optimization was considered to be secondary.

In the graphical database, each node N is represented by a record consisting of the following components:

- N.calls = list of nodes that N calls
- N.called = list of nodes that N is called from
- N.level = the level in the call hierarchy (non-negative integer)
- N.position = position in the process column (non-negative integer)
- N.indent = indentation within the level (non-negative integer)
- N.visit = flag indicating whether a node has been visited during traversal (boolean)
The algorithm works in two steps: first is node positioning and second is determination of the horizontal length of edge. The node positioning is done by topological sort [3] on relation "calls". After that, the nodes are ordered linearly in the column in such a way that each node is placed below all nodes that call it. Then all edges have the same direction, i.e. they point downwards. In Fortran programs, recursion is disallowed and hence the "calls" relations do not have cycles. Hence the topological sort always produces the linear ordering.

In the second step, the horizontal length of an edge is determined. The purpose of varying this length is to avoid overlapping between the edges. In order to explain the part of the algorithm, let us first consider, how the graph would be drawn in a 2-dimensional plane.

In a two-dimensional drawing of the call graph we displace nodes on the plane in two directions (Figure 2). A vertical displacement is needed in order to separate nodes that are called from other nodes higher in the call-graph hierarchy. We refer to this displacement as level. A node \( N \) with \( N\text{.called} = \emptyset \) is at the highest level.

A leaf node \( N \) in the hierarchy, i.e. with \( N\text{.calls} = \emptyset \), is at the lowest level. If \( N_i \) calls \( N_j \), that is \( N_j \in N_i\text{.calls} \), then \( N_i\text{.level} > N_j\text{.level} \). In figure 2, nodes \( a, b, d \) are at the highest level and node \( g \) is at the lowest level. A horizontal displacement is needed in order to distinguish between nodes that are at the same level. We refer to this displacement as indent.

Levels are computed in the following way:

1. (Initialization). For each node \( N \) in the column, assign \( N\text{.visit} := \text{FALSE} \) and \( N\text{.level} := 0 \).

2. For each node \( N \) for which \( N\text{.visit} = \text{FALSE} \), perform a depth first search through the "called" list. When a node \( N_i \) is reached for the first time from some node \( N_j \), set \( N_i\text{.visit} = \text{TRUE} \), and set its level to \( N_i\text{.level} = \max(N_i\text{.level}, N_j + 1) \). During the traversal record the maximum level reached so far. At the end of the traversal we invert the level values, so that the lowest level is zero.
Another value to be computed is the *spacing*, which is the distance between the vertical parts of the edges on the same level. Spacing in 2CG carries the same meaning as indent in two-dimensional layouts. There are three possible combinations between each pair of edges that need to be considered, and they are summarized in Figure 3. Case A is the situation where the vertical parts of two edges overlap vertically. Then, the vertical part of one edge must be spaced by one unit with respect to the other. Case B deals with the situation where the vertical parts of two edges do not overlap. Then their vertical parts can be drawn on the same vertical line. Finally, case C deals with another situation where two edges can be drawn on the same vertical line. This occurs when the two edges overlap but contain arrows directed towards the same nodes, i.e. their corresponding nodes have identical calls relations.

After all these values have been computed, the length of the horizontal part of each edge for a node N is determined:

\[
\text{length} := \text{N.level} \times \text{Lwidth} + \sum_{i=0}^{\text{N.level}-1} (\text{MaxIndent}(i) - \text{N.indent}) \times \text{Swidth}
\]

where, Lwidth and Swidth are units of length of a single level and a single spacing respectively. MaxIndent(i) is the total number of spacings for each level i.

Consider figure 4 which shows a possible 2CG translation of the graph in figure 2. Nodes a, b, d are all drawn within the same level, while nodes c, e are drawn in a distinct level. The pair of edges emanating from a and b demonstrate case A. The pair of edges emanating from b and c demonstrate case B. Finally, the pair of edges from c and e demonstrate case C.

In certain situations, the automatically laid-out graph is not the best one from the point of the user. In those situations, the user is allowed to preassign positions to some or all nodes in the column. The user can accomplish this by moving nodes to specific positions before invoking the layout algorithm. Then the topological sort will assign positions to all the nodes that do not have predetermined locations. In such a case, some edges may be bent upwards.

With this algorithm, the user can read all information which would be available in the two-dimensional display of the call graph.

References.

Figure 3. Spacing between edges