A Modern Graphic Flowchart Layout Tool

Sukhamay Kundu

Computer Science Dept, Louisiana State University
Baton Rouge, LA 70803, USA (`kundu@bit.csc.lsu.edu`)

Abstract. We describe the design of a flowchart layout tool for C-programs, which may contain break, continue, and return statements. We exploit the nesting relationship among the blocks to determine the positions of blocks and sub-blocks, which in turn determine the placement of connecting lines between the blocks. A special labeling technique is used to avoid the crossing of lines.

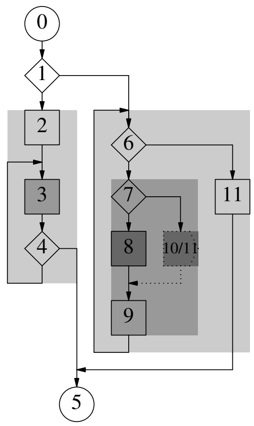
1. Introduction

A graphical display of a program's flowchart is helpful in understanding the program logic for beginning programmers. Flowcharts have many important uses, including static analyses (definition-use relationship of variables [1] and unreachable code segments [1, 2]), test-data generation and test-converge measures [2], computation of program slice [3, 4], and code optimization [1]. An important consideration in a flowchart-display is the clarity of showing the program blocks and their nesting structure. For large programs, it is also important to be able to form "summary display" by avoiding the details within selected blocks and to be able to expand those blocks when needed. We accomplish these using the nesting-tree of program blocks as the basis for creating the flowchart layout. The blocks with hidden details can be selected on the basis of their nesting levels, size (number of nodes in the block), nesting depth within the block, presence or absence of certain structures within them, etc. We avoid crossing among the connecting lines between nodes of a flowchart by using a special node-labeling scheme for the unstructured constructs involving break, continue, and return. Programs involving these constructs, but without goto's, maybe called *semi-structured*. The JAVAVIS tool [5] shows UML class diagrams (for object oriented programs) but not the flowchart. Our layouts are superior in many ways to those given by the commercial packages //www.fatesoft.com and //www.aisee.com. Flowchart layout is a special case of more general problem of graph drawing [6].

Fig. 1(i) shows the layout of a semi-structured flowchart; its input data, which is shown in Fig. 1(ii), is easily obtained by a simple preprocessing of the *C*-program text. Nodes and blocks with a deeper nesting-level are indicated by a darker shading; one can turn off the shading, if desired. The label 10/11 for the break-node 10 of the while-do loop shows that the original control (going to node 9 and indicated by a dotted line) is now modified going to node 11. In our flowchart layout tool, we first create an output file which describes the display in the "pic" language in UNIX. The actual display is then created by processing this file by the pic-utility in UNIX.

2. Construction of Layout

For our purpose, a *block* is a segment of a code which has single input point and a single output point when we disregard the unstructured control flows due to the



(i) The flowchart of a semi-structured program.

```
12 //numNodes
 0 start    1
 1 decision 2  6
 2 action   3
 3 action   4
 4 decision 3  5
 5 end     -1
 6 decision 7 11
 7 decision 8 10
 8 action   9
 9 action   6
10 break    9
11 action   5
```

(ii) The input data for the display in (i); each line after the first shows a node, its type, and its successor nodes. The nodes are numbered consecutively starting with 0 but otherwise arbitrarily and they may be listed in any order.

Figure 1. An example layout.

break, continue, and intermediate returns (other than that at the end of a function). The four basic block types are if-then-else (if-then being a special case), while-do loop (for-loop being a special case), do-while loop, and sequence.

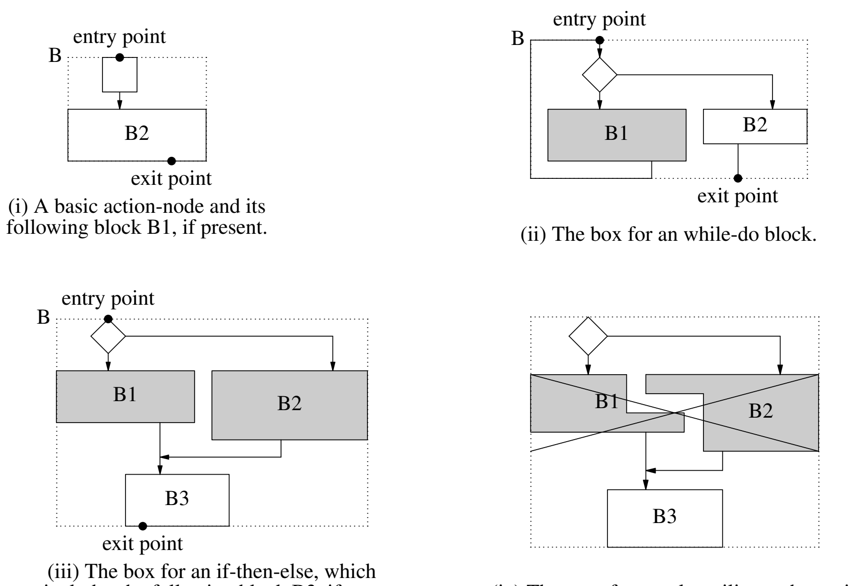
2.1. Notion of box

We view the layout as a hierarchy of boxes with horizontal and vertical sides. Each box has an entry point on its top side and an exit point on its bottom side; the box is connected to the outside world by a vertical line to and from there points, respectively. The box $B(x)$ associated with a node x in the flowchart contains x and extends just far enough on all sides to contain all its subboxes, including the connecting lines among them so that for each node $y \in B(x)$, y is reachable from x within $B(x)$ itself and $B(y) \subseteq B(x)$. The box corresponding to the end-node consists of only itself. $B(x)$ but does not contain nodes outside the innermost loop-body containing x, if any. The notion of a box should not be confused with a single entry and single exit block; a block is either contained in a box or disjoint from a box. We build the boxes in a bottom-up and inside-out fashion starting from the end-node.

Fig. 2(i) shows the box for an action node x, which includes x and the box B2 corresponding to the unique next node of x. Fig. 2(ii) shows the box for a white-do loop-test node, where B1 is the box for the loop-body and B2 is the box for the destination node of the false-branch at the loop-test node. There are few exceptions where B2 is not present (or equivalently, it is an empty box); for example, the while-do loop is the else-part of an if-then-else statement or it is the last block of another loop-body. Note that if the height of B2 is larger than the height of B1 plus the vertical separation between a box and an horizontal line, then exit-point(B2) = exit-point(B) and the vertical line (without the arrowhead) joining exit-point(B2) to exit-point(B) is not present. The box for a do-while loop-test node is similar to that of a while-do loop-test node. Fig. 2(iii) shows the box for an if-then-else test node, where B1 is the box for the then-part, B2 is the box for the else-part, and B3 is the box for the node that follows both then and else parts. B3 is empty when the if-then-else statement is the last block in a loop-body; B2 is empty for an if-then statement. We do not use rectilinear-shape boxes to optimize the layout space by packing the boxes horizontally or vertically as indicated in Fig. 2(iv) to avoid loss of visibility of the program blocks in a complex situation. A similar remark applies to Fig. 2(ii).

The vertical line segment to B2 in Fig. 2(i) do not always have the arrow-head; this would be the case, for example, if B2 is the box for an while-do loop because that would create two arrowheads on the line from B1 to the loop-test node in B2. Similar remark holds for the vertical lines to subboxes in Figs. 2(ii)-(iii). For each box B, we maintain the following parameters.

- (P1) width(B) and height(B),
- (P2) topDist(B) = the horizontal distance of the entry-point of B from its west-side and bottomDist(B) = the horizontal distance of the exit-point of B from its west-side, and
- (P3) needArrow(B) = yes, if the vertical entry line to B needs an arrowhead.



(i) A basic action-node and its following block B1, if present.

(ii) The box for an while-do block.

(iii) The box for an if-then-else, which includes the following block B3, if any.

(iv) The use of general rectilinear shapes is avoided to maintain the visibility of blocks.

Figure 2. The box-structure for sequence, while-do loop, and if-then-else.

2.2. Computation of box-parameters

For each case in Figs. 2(i)-(iii) and the do-while loop which is not shown there, we can compute the parameters (P1)-(P3) for a box B in terms of those for its component boxes; we can also determine the relative positions of the components within B. We show below the formulas for computing width(B), topDist(B), and bottomDist(B) for Fig. 2(iii), where we let topDist(B3) = bottomDist(B3) = width(B3) =

0 if B3 is not present. Let δ = bottomDist(B1) − topDist(B3) and HS = the unit vertical separation between two boxes.

$$\begin{array}{lrl}
(4.1) & \text{width(B)} & = \max\{\text{width(B1)} + \text{HS} + \text{width(B2)},\ \delta + \text{width(B3)}\},\ \text{if } \delta \geq 0 \\
 & & = \max\{-\delta + \text{width(B1)} + \text{HS} + \text{width(B2)},\ \text{width(B3)}\},\ \text{otherwise.} \\
(4.2) & \text{topDist(B)} & = \text{topDist(B1)},\ \text{if } \delta \geq 0 \text{ and} \\
 & & = -\delta + \text{topDist(B1)},\ \text{otherwise.} \\
(4.8) & \text{bottomDist(B)} & = \delta + \text{bottomDist(B3)},\ \text{if } \delta \geq 0 \text{ and} \\
 & & = \text{bottomDist(B3)},\ \text{otherwise.}
\end{array}$$

2.3. Computation of blocks

We determine the blocks from the depth-first tree [7] of the flowchart digraph, together with its back-arcs and cross-arcs, given in the form shown in Fig. 1(ii) and regarding the break, continue, and return statements as ordinary actions. Fig. 3(i) shows the depth-first tree for the input in Fig. 1(ii). A decision node N_j is a while-do loop-test node if there is a back-arc to N_j itself but to an ancestor of N_j prior to taking false-branch(N_j). The while-do loop-body consists of the nodes in the subtree of the depth-first tree at the child N_k of N_j, where true-branch(N_j) = (N_j, N_k). A decision node N_j is a do-while loop-test node if true-branch(N_j) is a back-arc to an ancestor node N_i of N_j; the loop-body then consists of all nodes in the subtree at N_i of the depth-first tree minus the nodes in the subtree at the end of the false-branch(N_j). In particular, the do-while loop-body may not be known when we backtrack from N_j (for example, if the loop-body contains if-then-else statements), but it will be known by the time we backtrack from N_i.

All other decision nodes N_j are if-then or if-then-else nodes. Suppose true-branch(N_j) = (N_j, N_k) and false-branch(N_j) = (N_j, N_n). N_j has no else-part if and only if false-branch(N_j) is a forward arc to a node N_m in the subtree of the depth-first tree at N_k. Otherwise, the subtree at N_k has a unique node N_m which is the head of all cross-arcs (≥ 1) leaving the nodes in the subtree at N_n. The subtree at N_m gives the nodes in B3 in Fig. 2(iii), the subtree at N_k minus the subtree at N_m gives the nodes in B1, and the subtree at N_n gives the nodes in B2. For $N_j = 1$ in Fig. 3(i), we have $N_k = 2$, $N_n = 6$, and $N_m = 5$; this gives B1 = {2, 3, 4}, B2 = {6, 7, 8, 9, 10, 11}, and B3 = {5}. Likewise, for $N_j = 7$, $N_k = 8$, $N_n = 10$, and $N_m = 9$; this gives B1 = {8} and B2 = {10}, and B3 = {9}.

2.4. Nesting-tree

The *nesting-tree*, which is an ordered tree, shows the nesting structure of the blocks by regarding the break, continue, and return statements as ordinary actions. Fig. 3(ii) shows the nesting-tree for the flowchart in Fig. 1(i). It has two types of nodes: the first type of nodes are the action nodes, it-test nodes (shown as circles with an inscribed diamond shape), while-do loop-test nodes (shown as double circles), and do-while loop-test nodes (shown as bold circles); the second type of nodes are the place holder nodes for then- and else-parts of an if-test and the root node; they are shown as shaded nodes without labels. For an if-then statement, the second child of the if-test node has no children; the do-while loop-test node also has no

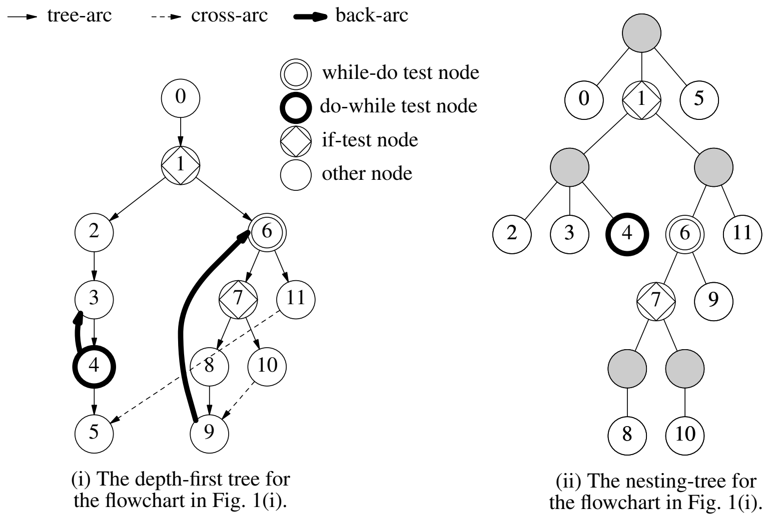


Figure 3. The depth-first tree and nesting-tree for the flowcharts in Fig. 1(i).

children. The while-do loop-test node may have an arbitrary number (≥ 1) of children, which correspond to the blocks in the loop-body. The children of a node correspond to a sequence of single input single output blocks, with one exception. The blocks in a do-while loop-body appear as left brothers preceding the loop-test node; the starting block of the body is given by the node which is head of the back-arc (in the depth-first tree) from the loop-test node. See Fig. 3(ii). (We do not make these brother nodes the children of the do-while loop-test node to allow the construction of the nesting-tree with a minimal change to the depth-first tree of the flowchart as we traverse the latter, although this could be done; recall that the do-while loop body may not be completely known if it contains if-then-else nodes at the time we process the back-arc.) The children of the root show the sequential blocks in the program.

Construction of the nesting-tree is straightforward from the results of the depth-first processing. Since the number of links in the flowchart digraph $\leq$ twice the number of nodes, we get the following theorem.

Theorem 1. The generation of nesting-tree takes $O(N)$ time where N = #(nodes in the flowchart).

Note that the layout in Fig. 1(i) cannot be obtained in a depth-first fashion from the flowchart digraph in Fig. 3(i). For example, the vertical spacing between nodes 4 and 5 cannot be determined until the whole digraph is processed. Likewise, the horizontal space between nodes 2 and 6 cannot be determined at the time of visiting node 6, but only after we know the body of the while-do loop. The creation of a summary display to hide the details within a block means for the most part ignoring the subtree of a node corresponding to that block, with a slightly different processing for do-while loops because of the particular way it is represented in nesting-tree.

2.5. Generation of pic-code for layout

Figs. 4(i)-(ii) show a simple flowchart layout and its pic-code generated by our tool. For each node *k* in the flowchart, there are two objects Nk and Bk in the pic-code, where Nk is a small circle or square or diamond (depending on the type of node *k*) for the display of node *k* itself and Bk is an auxiliary box associated with node *k* (as described in Section 2.1) for controlling the positioning of the nodes within that box. As we process node *k* in the nesting-tree in the bottom-up fashion, the block of lines of pic-code indicated in Fig. 4(ii) on the leftside margin starting with the line "`Nk: box ...`" is created by piecing together the pic-code segments generated previously for subtrees at node *k*. Although the specification of Nk itself refers to Bk, the pic-code "`Bk: box wid ...`" is generated only when we create the block of pic-code for the parent box of Bk. The first three lines in Fig. 4(ii) is the same for all flowcharts; here, hu = horizontal spacing unit and vu = vertical spacing units. These lines of pic-code are created last along with the next two lines; in particular, B0 is always placed at position (0,0). The pic-code for N6 is determined here first. Our layout-tool generates the pic-code that is simple and directly related to the structure of the display, and is not optimized as such. This is done to simplify the debugging of our tool and to allow easy manual modification for changing the display if such a need should arise. An extreme example of non-optimization of the pic-code can be seen in parts like "`f*(0+0)`", which could be simplified to "`0`". Other examples are the computations of box-width (w) and box-height (h), based on formulas in Section 2.2, before the pic-code line for each Bk.

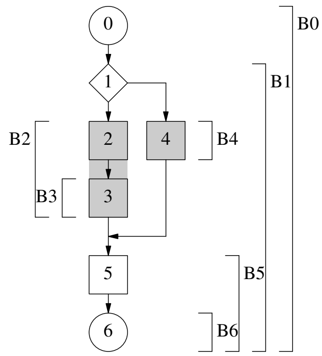
3. Conclusion

We have described a simple and efficient technique for producing an elegant layout of the flowchart of a semi-structured *C*-programs, including the choice of hiding the detials within selected blocks. We assume there are no goto's in the *C*-program because of the difficulties in defining a suitable notion of a block which is meaningful in terms of program logic (block hierarchy) and hence the layout. Our approach applies equally well to programs in other block-structured languages.

Reference

1. Aho, A. and Ullman, J.D., *Principles of Compiler Design,* Addison-Wesley Publ. Co., Mass., 1977.
2. Beizer, B. *Software testing techniques,* Van Nostrand Reinhold Comp., 1983.
3. Weiser, M., Program slicing, *IEEE Trans. Soft. Engg,* 10(1984), pp. 352-357.
4. Horwitz, S., Reps, T., and Binkley, D., Interprocedural slicing using dependence graphs, "*ACM Transactions on Programming Languages and Systems 12(1990), pp. 26-60.*
5. Rainer, O. and Thomas, S., JAVAVIS: automatic program visualization with object and sequence diagrams using the Java Debug Interface (S. Diehl, ed.) in *Software Visualization,* LNCS #2269, pp. 176-190, 2002.
6. Battista, G.Di., Eades, P., Tamassia, R., and Tollis, I.G., *Graph drawing,* Precntice Halls, 1999.

7. Coreman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to algorithms,* The MIT Press, 2001.



(i) A simple flowchart; the boxes are identified in the order B6, B5, B3, B2, B4, B1, and B0.

```
hu=0.10; vu=0.10; bw=2*hu; bh=2*vu; hs=4*hu; vs=3*vu; f=0.1 #fill-factor
boxwid=bw; boxht=bh; ellipsewid=bw; ellipseht=bh
arrowht=vu/2; arrowwid=arrowht/2
w=5; h=18
B0: box wid w*hu ht h*vu at (0,0) invis filled f*0
N0: ellipse "0" with .n at B0.nw+(1*hu,0)
w = 1+4; h=15
B1: box wid w*hu ht h*vu with .nw at N0.n-(1*hu,vs) invis filled f*(0+0)
N1: box "1" with .n at B1.nw+(1*hu,0) invis
line from N1.n to N1.e to N1.s to N1.w to N1.n
w=1+1; h=5
B2: box wid w*hu ht h*vu with .nw at N1.n-(1*hu,vs) invis filled f*(1+0)
N2: box "2" with .n at B2.nw+(1*hu,0)
w = 1+1; h=2
B3: box wid w*hu ht h*vu with .nw at N2.n-(1*hu,vs) invis filled f*(1+0)
N3: box "3" with .n at B3.nw+(1*hu,0)
line -> from N2.s to B3.nw+(1*hu,0)
line -> from N1.s to B2.nw+(1*hu,0)
w=1+1; h=2
B4: box wid w*hu ht h*vu with .nw at B2.ne+(hu,0) invis filled f*(1+0)
N4: box "4" with .n at B4.nw+(1*hu,0)
line -> from N1.e to (N4.x,N1.e.y) to N4.n
x=B2.sw.x+1*hu; y=B2.sw.y; y2=y-vu
L: line from (x,y) to (x,y2)
line -> from B4.sw+(1*hu,0) to (B4.sw.x+1*hu,y2) to (x,y2)
w=1+1; h=5
B5: box wid w*hu ht h*vu with .nw at L.s-(1*hu,vu) invis filled f*(0+0)
line -> from L.s to L.s-(0,vu)
N5: box "5" with .n at B5.nw+(1*hu,0)
w = 1+1; h=2
B6: box wid w*hu ht h*vu with .nw at N5.n-(1*hu,vs) invis filled f*(0+0)
N6: ellipse "6" with .n at B6.nw+(1*hu,0)
line -> from N5.s to B6.nw+(1*hu,0)
line -> from N0.s to B1.nw+(1*hu,0)
```

(ii) The groupings of pic-code for the various boxes in (i).

Figure 4. Illustration of the pic-code generated by our tool.