Generalized structured programs and loop trees

Ward Douglas Maurer*

Computer Science Department, George Washington University, Washington, DC 20052, USA

Received 22 September 2005; received in revised form 1 November 2006; accepted 21 February 2007
Available online 13 April 2007

Abstract

Any directed graph, even a flow graph representing “spaghetti code”, is shown here to have at least one loop tree, which is a structure of loops within loops in which no loops overlap. The nodes of the graph may be rearranged in such a way that, with respect to their new order, every edge proceeds in the forward direction except for the loopbacks. Here a loopback goes from somewhere in a loop \( L \) to the head of \( L \). We refer to such a rearrangement as a generalized structured program, in which forward goto statements remain unrestricted. Like a min-heap or a max-heap, a loop tree has an array representation, without pointers; it may be constructed in time no worse than \( O(n^2) \) for any program written in this fashion. A scalable version of this construction uses a label graph, whose only nodes are the labels of the given program. A graph has a unique loop tree if and only if it is reducible.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Generalized structured program; GSP; Loop trees; Reducible graphs; Overlapping loops; Strongly connected components

1. Introduction

There exist today many variants on the idea of a structured program. Böhm and Jacopini [8] consider programs which are decomposable into \( \Pi, \Phi, \) and \( \Delta \) (in other words, sequencing, if-then-else, and while loops); these programs contain no goto logic at all, except what is implied by the constructions \( \Phi \) and \( \Delta \). In what follows, we refer to these as well-structured programs. The basic elements of a structured program have then been expanded to include break and continue statements (sometimes called leave and cycle); labeled break and labeled continue; loops with an else clause (executed only on normal, not abnormal, exit from a loop); and the like.

Our purpose here is to consider an even more general kind of structured program, which we call a generalized structured program, or GSP. Such programs are built up from sequencing, if-then-else, and loops of various kinds, including (possibly labeled) continue statements; and they may also contain goto statements in the forward direction. Any GSP, then, is an ordered set of statements \( P_1, P_2, \ldots, P_n \), with a goto statement from \( P_i \) to \( P_j \) being a forward goto if \( i < j \). A backward goto in a GSP arises from the end of a loop, or a continue statement in a loop, and it always goes back to the head of the loop (in a for loop, the head follows the initialization).

It should be clear that GSPs are more general than other kinds of structured programs. A GSP can contain a goto statement which jumps out of several loops at once, or which jumps into several loops at once, or which does both at the same time. Indeed, even in the absence of loops, a program can contain forward goto logic without being decomposable into sequencing and if-then-else.

* Tel.: +1 202 994 5921; fax: +1 202 994 4875.
E-mail address: maurer@gwu.edu.

0167-6423/S - see front matter © 2007 Elsevier B.V. All rights reserved.
We submit that the study of GSPs is useful for two reasons. First, the GSP represents the widest possible generalization of a structured program without losing its structure. If even one backward goto is added to a structured program \( P \), it introduces new structure to \( P \) unless it goes back to the head of a loop \( L \) from somewhere within \( L \). A forward goto by itself, however, introduces no new looping logic.

Second, and more importantly, as shown in Section 7 below, every unstructured program \( P \) is isomorphic to a generalized structured program \( P' \). Informally, we may say that “there is no such thing as spaghetti code”, or that “all code is structured” (if by “structured” we mean generalized structured, as defined above). Note carefully that we do not mean that \( P \) is merely equivalent to \( P' \); the correspondence is an actual graph isomorphism at the flow graph level. The nodes of the flow graph of \( P' \) are exactly those of the flow graph of \( P \), with no variables or nodes added and none deleted; only the order of the nodes is affected, and the new order may be found constructively. Indeed, we have written an application of this result to assembly language programs [28]; this takes, as input, an unstructured program, written in a certain order, and rearranges it into a new order, in such a way that it is clearly a GSP (see Section 9.3 below).

This result has the following applications and further consequences:

- It provides a complementary extension to the result of Böhm and Jacopini. Using their terminology, every unstructured program is proved in [8] to be weakly decomposable into a well-structured program, whereas our result says that every unstructured program is strongly decomposable into a GSP. Strong decomposability, in this sense, is the same as graph isomorphism. Note that not every unstructured program is decomposable (either strongly, or using an intermediate notion of decomposability) into a well-structured program, according to [8].
- It suggests a method of rearranging programs written in languages, resulting in a GSP. Since isomorphism is at the flow graph level, not the program level, there can be minor syntactic differences between a program and its rearrangement, even for assembly language programs. Further syntactic differences are found when a compiled program is rearranged, if, indeed, it can be rearranged at all; this sometimes requires it to be written in a language which allows the goto. This subject is taken up further in Sections 9.3 and 9.4 below. Even with syntactic differences, however, the flow graphs of the original program and the rearranged program are still isomorphic.
- It motivates research into the analysis of GSPs. A well-structured program may be analyzed by any of a wide variety of methods, such as those of Hoare’s logic (see [6], for example). It is easy to conjecture that GSPs may be analyzed by extending those methods, although this remains an open question at the time of writing. Forward goto statements, as noted above, do not, of themselves, introduce new looping logic to a program. In particular, an innermost while loop, even if it contains forward goto statements, can be analyzed by repeated back substitution.
- It solves the problem of overlapping loops. Previous works on unraveling unstructured programs [31,41] take up overlapping loops as a separate case, and involve transformations which do not always correspond to graph isomorphisms. Our main result here, however, shows that every flow graph of an unstructured program is isomorphic to the flow graph of a GSP, in which loops form a hierarchical structure in which no loops overlap. In fact, overlapping loops are an illusion; an example, illustrating this, is given at the end of Section 2 below.
- Because our result is about general directed graphs, it also applies to other kinds of such graphs, such as call graphs. In [29] we have used this application in improving the speed of object code when the source language allows nested procedures; a specific application of this to the Ada language appears in [30]. This work extends an earlier technique which we developed, but which works only in the absence of recursion, when the call graph is acyclic. The call graph is then set up like a project graph in the AOV (Activity On Vertex) version of CPM (Critical Path Method) [21, pp. 310–315]; such a graph is required to be acyclic. Every variable in every procedure then receives a single fixed offset, for the lifetime of the project, so that stack adjustment on procedure call and return is eliminated, and only one base register is needed. If the call graph is not acyclic, we use the results of this paper in determining what stack adjustments, and what base registers, are actually needed (normally many fewer than with classical methods). Further details are given in [29] and [30].

2. Loop trees

Every well-structured program \( P \) has what may be called a syntactic loop tree, in which the root represents \( P \) and all other nodes represent loops in \( P \). Descendants of \( X \) in \( P \) represent loops in \( P \) contained in \( X \). Fig. 1 gives an example (using for loops rather than while loops).
void matrixmult() { // program P
    for (i = 1; i <= n; i++) // loop A
        for (j = 1; j <= n; j++) // loop B
            input(a[i][j]);
    for (i = 1; i <= n; i++) // loop C
        for (j = 1; j <= n; j++) // loop D
            input(b[i][j]);
    for (i = 1; i <= n; i++) // loop E
        for (j = 1; j <= n; j++) // loop F
            sum = 0;
        for (k = 1; k <= n; k++) // loop G
            sum += a[i][k]*b[k][j];
            c[i][j] = sum;
    } 
} 

Fig. 1. A program containing several loops, and its syntactic loop tree.

Here the outer loops are A, C, E, and H, while the inner loops are B (contained in A), D (inside C), F (inside E), G (inside F), and J (inside H). Therefore, A, C, E, and H are children of P in the loop tree, while B is a child of A; D is a child of C; F is a child of E; G is a child of F; and J is a child of H.

void matrixmult2() { // program Q
    for (i = 1; i <= n; i++) // loop E
        for (j = 1; j <= n; j++) // loop F
            sum = 0;
        for (k = 1; k <= n; k++) // loop G
            sum += a[i][k]*b[k][j];
            c[i][j] = sum;
    } 

Fig. 2. A program which is a single loop, and its syntactic loop tree.

If the entire program Q is a single loop E, it occurs twice in the loop tree, once as Q and once as E (note that the entire program is always the root of the tree, whether or not it is a loop). This is illustrated in Fig. 2, which shows part of the program of Fig. 1, including the loops E, F, and G. We refer to this as a syntactic loop tree because its form follows from the syntax of the program. Suppose that we now translate the program Q of Fig. 2 into its unstructured equivalent, with goto statements, as in Fig. 3. This figure also shows line numbers at the left, and the flow graph of the resulting program (see Definition 1 below), labeled by these line numbers.

Here Q, arguably, still has a loop tree, but it does not quite represent the same loops as before. In particular, as we saw, E and Q are the same, in Fig. 2. However, in Fig. 3, we are not including the initialization i = 1; in the loop E. Here we have the semantic loop tree, which is concerned, as its name implies, with the meaning of a loop, rather than its form. In Fig. 2, i = 1; is part of the syntax of the loop E, and so it is included in the syntactic loop tree. In Fig. 3, it is not included in the semantic loop tree, because it is not meaningfully part of the loop E; one cannot “loop back” to it from anywhere inside that loop.

The precise definition of a semantic loop tree, in this sense, is given in Definition 9 below. Our purpose here is to show that every unstructured program has a semantic loop tree. No matter how many goto statements a program contains, its flow graph may be rearranged in such a way as to show a structure of loops within loops. This graph may be reordered in such a way that, with respect to the new order, all control passes in the forward direction except for the loopbacks. Here a loopback is an edge from somewhere in a loop L to the head of L. This is proved in Theorem 1.
void matrixmult2() { // program Q
    i = 1;
    E: if (i > n) goto EX;
    j = 1;
    F: if (j > n) goto FX;
    sum = 0;
    k = 1;
    G: if (k > n) goto GX;
    sum += a[i][k]*b[k][j];
    k ++;
    goto G;
    GX: c[i][j] = sum;
    j ++;
    F: goto F;
    FX: i ++;
    goto E;
  }

Fig. 3. An unstructured program, its semantic loop tree, and its flow graph.

As an example, consider the flow graph of Fig. 4. This is a simplified version of a variant of depth-first search, translated into assembly language and rearranged as a GSP in [28]. The graph of Fig. 4 appears to have no structure; indeed, it appears to have overlapping loops, P2-P3-P4-P5 and P5-P6-P10-P11. The reader is invited to attempt to find a rearrangement of this flow graph, involving a structure of loops within loops in which no loops overlap, accompanied by forward goto statements. (The solution, together with an explanation of its derivation, is given in Section 8.1 below.)

3. Notation and basic lemmas

In what follows, by a graph we shall always mean a finite directed graph, possibly with self-loops, but without parallel (that is, multiple) edges. We use the following standard terms for graphs: node (=vertex), edge, path, simple path, cycle, acyclic graph, dag, reachable, subgraph, induced subgraph, self-loop. These are defined in standard texts on graph theory.
Definition 1. The flow graph of a program $P$ is a graph whose nodes are the executable statements of $P$, and in which an edge from $U$ to $V$ denotes the fact that $V$ is, or can be, the next statement to be executed after $U$. This definition may sometimes be extended by associating, with each node in the flow graph, a group of statements (under restrictive conditions), rather than a single statement; see Definition 13 below.

Definition 2. A graph $G$ is rooted if it has a node, called its start node, from which every other node in $G$ is reachable. If $G$ is a flow graph, it may be assumed to be rooted, since any nodes not reachable from the start node may be removed from $G$ without changing its computational behavior. Coincidentally, our main theorem is proved, initially, for rooted graphs, although in Section 12 below we generalize it to all graphs.

Definition 3. A strong component of a graph $G$ is the induced subgraph of an equivalence class of the nodes of $G$ under the relation $R$, with $XRY$ meaning that there are paths in $G$ from $X$ to $Y$ and from $Y$ to $X$. More generally, a subgraph $H$ of a graph $G$ is strongly connected if, for any two nodes $X$ and $Y$ in $H$, there is a path (possibly of length zero) from $X$ to $Y$.

Under this definition, a strong component might consist of a single node and no edges. We sometimes need to exclude this case, as follows.

Definition 4. A strong component of a graph is trivial if it has no edges (and therefore exactly one node). Otherwise, it is non-trivial.

Definition 5. The component graph of a graph $G$ is the graph $G'$ whose nodes are the (trivial and non-trivial) strong components of $G$, and in which an edge from $U$ to $V$ in $G'$, where $U \neq V$, denotes at least one edge in $G$ from a node in $U$ to a node in $V$. The start node of $G'$ is the strong component of $G$ which contains its start node.

If the condition $U \neq V$ were omitted, any edge in $G$ from a node in $U$ to another node in $U$ would give rise to a self-loop in $G'$ from $U$ to itself. The above condition eliminates such self-loops; indeed (see Lemma 4 below), $G'$ contains no loops at all.

Definition 6. A topological sort of a graph $G$ is an arrangement of its nodes as $P_1, \ldots, P_n$, such that, for every edge $P_i \rightarrow P_j$ in $G$, we have $i < j$.

The following lemmas are standard; their proofs are omitted.

Lemma 1. A strong component $L$ of a graph $G$ is maximal strongly connected. That is, there cannot exist a strongly connected subgraph of $G$ which properly contains $L$.

Lemma 2. The union of two strongly connected subgraphs of a graph $G$ is strongly connected if they have a non-void intersection.

Lemma 3. Every cycle in a graph is contained in some strong component of it.

Lemma 4. The component graph of $G$, as in Definition 5, is acyclic.

Lemma 5. A graph has a topological sort $P_1, \ldots, P_n$ if and only if it is acyclic. If it is acyclic, any node of indegree 0 may be taken as $P_1$ here.

Lemma 6. If there is a path from node $U$ to node $V$ in a graph $G$, then there is a simple path from $U$ to $V$ in $G$ (that is, one with no repeated nodes).

Lemma 7. Let $N$ be a node in a graph $G$, contained in a non-trivial strong component $L$ of $G$. Then $L$ is induced by the set of all nodes $K$ such that there is a cycle in $G$, of non-zero length, which contains both $N$ and $K$.

4. Outer loops and strong components

We have seen that semantic and syntactic loops are not always the same. In Section 4.3 below, we will argue that the semantic outer loops of a flow graph are precisely its non-trivial strong components (see Definitions 3 and 4 above). Before we do this, however, we take up some further differences between syntactic and semantic loops besides the initialization of a for loop, as mentioned in Section 2 above.
4.1. In the semantic loop, but not in the syntactic loop

Our first example goes back to FORTRAN; it is forbidden in many other programming languages, but it is allowed in C, and we present a C version of it in Fig. 5(a). This code sets $\text{max}$ equal to the maximum of $t[1]$ through $t[n]$, and sets $\text{index}$ in such a way that $\text{max} = t[\text{index}]$ at the end. In C, a for loop such as this is allowed to be written as in Fig. 5(b). In that figure, the statements $\text{max} = t[i]$; and $\text{index} = i$; are outside the for loop; however, their meaning, in an informal sense, is that of statements which are inside the loop.

We would like, therefore, to regard these statements as being inside the corresponding semantic loop, even though they are outside the syntactic loop. Thus, in Fig. 5(b):

- The syntactic loop contains lines 3b, 4b, 5b, and 6b (only).
- The semantic loop contains line 3b (except for $i = 2$; which is not in the semantic loop, as we saw in Section 2 above). The semantic loop also contains lines 4b, 5b, 6b, 8b, 9b, and 10b.

4.2. In the syntactic loop, but not in the semantic loop

Our second example, presented in C in Fig. 6, is allowed in several programming languages. If we look merely at the structure of the program of Fig. 6(a), without thinking about what it does, it appears to be two nested loops. Now, however, let us replace the “inner loop” by a goto statement which goes to that loop, as in Fig. 6(b). Here we obviously have two sequential loops, not two nested loops. In fact, this program returns true if $x$ is equal to some $t[i]$, for $1 \leq i \leq n$, and also equal to some $u[j]$, for $1 \leq j \leq n$ (possibly with $i \neq j$); and it returns false otherwise.
The programs of Fig. 6(a) and (b) are exactly the same, with respect to the ordering of their statements during execution (other than the goto statements). The inner loop in Fig. 6(a) is, therefore, not contained, in any meaningful way, in the outer loop. We would like, therefore, to regard the statements of this inner loop as being outside the outer semantic loop, even though they are inside the outer syntactic loop. Thus, in Fig. 6(a):

- The outer syntactic loop contains lines 2a, 3a, 4a, 5a, 6a, and 7a.
- The outer semantic loop contains line 2a (except for \( i = 1 \); which is not in the semantic loop, as we saw in Section 2 above). The semantic loop also contains lines 3a and 7a, but not lines 4a, 5a, or 6a.

### 4.3. The meaning of a semantic loop

In the discussion below, by a loop we always mean a semantic loop. We now consider how to set up a formal definition of an outer loop; inner loops will be taken up in Section 6 below. We first consider the case in which a loop has one and only one head \( H \); the general case is taken up in Section 4.4 below. We denote by \( L \) the outer loop containing \( H \). In order for a statement \( V \) to be in \( L \), there must be a (directed) path from \( H \) to \( V \), in the graph, and also from \( V \) back to \( H \). Thus, in our three examples:

- In Section 2 above, we spoke of the initialization \( I \) of a for loop as not being in the semantic loop \( L \). Here \( H \) is the statement which immediately follows \( I \); and \( I \) is not in \( L \), because there is no path from \( H \) back to \( I \) in the graph (although there is one from \( I \) to \( H \)).
- In Section 4.1 above, we spoke of statements 8b, 9b, and 10b as being in the semantic loop. Here \( H \) is in statement 3b. There is a path from \( H \), through statement 4b, to statement 8b; from there to statements 9b and 10b; from there to statement 5b; and from there (because this ends the for loop), back to \( H \).
- In Section 4.2 above, we spoke of statements 4a, 5a, and 6a as not being in the semantic loop. Here \( H \) is in statement 2a. There is a path from \( H \), through statement 3a, to statement 4a, and from there to statements 5a and 6a. However, there is no path from statement 4a, 5a, or 6a back to \( H \); every path from any of these statements goes to either return true or return false.

Now consider the set of all statements \( V \) satisfying the above condition; that is, for every such \( V \), there are paths in the flow graph \( G \) from \( H \) to \( V \) and from \( V \) to \( H \). This set already has a mathematical name; it is the strong component of \( G \) that contains \( H \), as in Definition 3 above.

### 4.4. Multiple entry points and trivial strong components

Let \( L \) be a strong component of a graph \( G \). If \( H \) is an entry point of \( L \), then, given a statement \( V \) in \( L \), there are paths in \( L \) from \( H \) to \( V \) and from \( V \) to \( H \). If \( K \) is a second entry point of \( L \), however, then, since \( L \) is strongly connected, there are also paths in \( L \) from \( K \) to \( V \) and from \( V \) to \( K \). The same is true of any further entry points of \( L \); so the identification of \( L \) with an outer loop is independent of how many entry points \( L \) has.

Any strong component \( L \) of \( G \) may thus be considered as an outer loop of \( G \), provided that \( L \) has at least one edge. If the statement \( V \) is not in any outer loop of \( G \), then \( V \), by itself, is a trivial strong component of \( G \), in the sense of Definition 4 above. Clearly such a strong component must have exactly one node; it cannot have more, since then it would not be strongly connected. On the other hand, a non-trivial strong component might have just one node \( V \), with an edge from \( V \) to itself. We can now define (semantic) outer loops: an outer loop of a graph is a non-trivial strong component of it (see Definition 9 below).

### 5. Loop heads and entry points

In the discussion above, we have used the terms “head of a loop” and “entry point” without giving formal definitions for them. The definition of an entry point is immediate.

**Definition 7.** An entry point of a subgraph \( G' \) of \( G \) is a vertex \( Y \) in \( G' \) with an edge to \( Y \) from some vertex \( X \) in \( G \) that is not in \( G' \).

The definition of the head of a loop depends on a lemma.
Lemma 8. A strong component $L$ of a rooted graph $G$ contains one or more entry points if and only if it does not contain the start node $S$ of $G$.

**Proof.** Suppose that $L$ contains $S$; we show that $L$ contains no entry point $Y$ of $G$. Suppose the contrary; then $L$ contains $Y$, and, by Definition 7, there is an edge from $X$ to $Y$ where $X$ is some node of $G$ which is not in $L$. Since $S$ is the start node, there is a path from $S$ to $X$, by Definition 2. Since $L$ is strongly connected, there is a path from $Y$ to $S$, by Definition 3; and hence $X$ is in a cycle that goes from $S$ through $X$ to $Y$ and then back to $S$. The union of this cycle with $L$ is strongly connected, by Lemma 2; and this contradicts our assumption that $L$ is a strong component and therefore, by Lemma 1, a maximal strongly connected subgraph of $G$.

Conversely, suppose that $L$ does not contain $S$. Let $W$ be a node in $L$; since $G$ is rooted, there is a path $\pi$ in $G$ from $S$ to $W$, by Definition 2. Let $X$ be the last node on $\pi$ that is not in $L$. Clearly $X$ exists, since $S$ is not in $L$; and $X$ is not $W$, which is in $L$. Therefore there is a node $Y$ in $\pi$ which immediately follows $X$. Since $Y$ is in $L$, while $X$ is not, $Y$ is an entry point of $L$, by Definition 7. □

Definition 8. A head of a strong component $L$ of a rooted graph $G$ is a node in $L$ which is either an entry point of $L$ or the start node of $G$.

Lemma 8 then shows that $L$ must contain at least one head. In a reducible graph (see Definition 10 below), $L$ always contains exactly one head.

Two loops cannot have the same head. This may seem counterintuitive; thus in the flow graph of Fig. 7, there appear to be two loops, $\{P3, P4, P5\}$ and $\{P3, P6, P7\}$, both with the head $P3$. However, this is the flow graph of the program of Fig. 7 (a well-known implementation of Euclid’s algorithm, setting $i = \gcd(m, n)$); and here there is clearly only one loop. This illustrates the general pattern: whenever there appear to be several loops with the same head, there is actually if-then-else logic which can lead to each of these “loops”, so that there is really just one loop with that head.

6. Inner loops

A loop $L2$ in a graph $G$, inner to a loop $L$ with head $H$, cannot contain any edges which lead to $H$ from other nodes in $L$. Suppose the contrary, so that $L2$ contains an edge leading to $H$, which is, in particular, contained in $L2$. If $H$ is the start node of $G$, it is also the head of $L2$. Otherwise, $H$ is an entry point of $L$, by Definition 8, so there is an edge to $H$ from $X$, which is not in $L$. Since $L2$ is contained in $L$, $X$ is also not in $L2$; therefore $H$ is a possible head for $L2$. But $H$ is also the head of $L$, contradicting our statement that two loops cannot have the same head.

Now consider the graph $B$, which is the graph $L$ after removing all edges which lead to $H$ from other nodes in $L$. By what we have just seen, a loop $L2$, inner to $L$, must be contained in $B$. But an outer loop of $B$ is a non-trivial strong component of $B$, by the same logic as in Sections 4.3 and 4.4 above. That then becomes a first-level inner loop of $L$; and this process can now be iterated, producing all the loops in the original graph. In order to make this precise, we define several concepts as follows.

Definition 9. A non-trivial strong component $L$ of a graph $G$ is an outer loop of $G$, or a loop at level 1 of $G$. Given a choice of head $H$ for $L$ as in Definition 8:
(a) The body $B$ of $L$ is $L$ with all its loopbacks removed, where a loopback of $L$ is an edge to $H$ from some other node in $L$.
(b) A loop at level $n$ of $G$ is a loop at level $n - 1$ of the body of some outer loop of $G$.
(c) A semantic loop tree $T$ of $L$ has $L$ as its root. Each outer loop $L'$ of the body $B$ of $L$ is a child of $L$ in $T$.
The subtree of $T$ having $L'$ as its root is a semantic loop tree of $L'$. The loopbacks of $T$ are the loopbacks of $L$ together with all loopbacks of all children of $L$ in $T$.
(d) A semantic loop tree $T$ of $G$ has $G$ as its root. Each outer loop $L$ of $G$ is a child of $G$ in $T$. The subtree of $T$ having $L$ as its root is a semantic loop tree of $L$. The loopbacks of $T$ are the loopbacks of all children of $G$ in $T$.

In what follows, by “loop tree” we always mean a semantic loop tree. The term loopback is meant to suggest that these are ways of “looping back” to the head of a loop.

**Lemma 9.** The head of a semantic loop $L$, by itself, is a trivial strong component of its body $B$.

**Proof.** The head $H$ of $L$ cannot be contained in any larger strong component of $B$, since there are no edges in $B$ that lead to $H$ (since these were all removed from $L$ to form $B$), by Definition 9(a).

**Lemma 10.** The body $B$ of a semantic loop $L$ is a rooted graph, with the head $H$ of $L$ being the start node of $B$.

**Proof.** Since $L$ is strongly connected, there is a path in $L$ from $H$ to $V$, for any node $V$ in $L$, by Definition 3. By Lemma 6, there is a simple path $\pi$ in $L$ from $H$ to $V$. Here $\pi$ cannot contain any edges of $L$ that are not in $B$, because all these edges lead to $H$ and thus $H$ would be contained twice in the path, contradicting the definition of a simple path. Therefore $\pi$ is entirely in $B$, and $V$ is reachable from $H$ in $B$. The proof now follows from Definition 2.

### 7. The main theorem

Definition 9 implies that every rooted graph – and, as we show in Section 12, every graph whatsoever – has at least one (semantic) loop tree. From this, we may prove the basic version of our fundamental theorem: every rooted graph may be reordering in such a way that, with respect to the new order, the only reverse edges are the loopbacks. This may be done by removing all loopbacks, at all levels, from a graph $G$; it is then not difficult to show that the resulting graph must be acyclic. By Lemma 5, it therefore has a topological sort, and this gives the required rearranged order of $G$. Rather than formalizing this argument, we proceed immediately to a stronger version of the theorem, which will be useful in developing a compact representation for loop trees (see Section 9.1 below).

**Theorem 1.** Given a rooted graph $G$ with $n$ nodes, and with a loop tree $T$, there exists an ordering $P_1, \ldots, P_n$ of the nodes of $G$, such that:

(a) For every edge $E = P_i \rightarrow P_j$ in $G$, we have $i < j$ unless $E$ is a loopback of $T$.
(b) If $L$ is a loop in $T$ with $k$ nodes, and with head $P_i$, the nodes of $L$ are precisely $P_i, P_{i+1}, \ldots, P_{i+k-1}$.

**Proof.** We use the notation $x(X, T)$, where $X$ is a node in $G$, to denote the index of $X$ in the ordering $P_1, \ldots, P_n$ (in other words, $x(P_j, T) = j$). We determine $x(X, T)$ for all $X$ in $G$, using induction on the height $h$ of $T$. If $h = 0$, then $G$ is acyclic, and therefore has a topological sort by Lemma 5. This is then the required ordering of the nodes of $G$; and (a) and (b) above follow immediately, since there are no loops, and therefore no loopbacks.

Now assume that $h > 0$, so there is at least one loop $L$ in $T$, which is a non-trivial strong component of $G$. The component graph $G'$ of $G$, as in Definition 5, is acyclic, by Lemma 4, and therefore has a topological sort, by Lemma 5. This is an ordering $Q_1, \ldots, Q_m$ of the nodes of $G'$, which are the (trivial and non-trivial) strong components of $G$. Every node $X$ in $G$ is in one of the $Q_k$, and, in this case, we define $c_X$ to be $k$. We also define $n_k$ to be the number of nodes in $Q_k$; and, for $0 \leq i \leq m$, we define $n'_i = \sum_{j=1}^{i} n_j$. In particular, $n'_0 = 0; n'_m = n; n'_i = n'_{i-1} + n_i$ for $1 \leq i \leq m$; and $n'_a \leq n'_b$ whenever $0 \leq a \leq b \leq m$.

Here $L = Q_k$ has a loop tree $T'$ which is the subtree of $T$ with root $Q_k$, and whose height $h'$, therefore, is less than $h$. By inductive assumption, $x(X, T')$ is already defined. We now recursively define

$$x(X, T) = n'_i x_{i+1} + x(X, T').$$  \hspace{1cm} (1)

We first show that the $x(X, T)$ so defined are, in fact, all the indices from 1 through $n$. This is clear, by Definition 6, if $h = 0$; if $h > 0$, we show, more generally, that the $x(X, T)$ for all $X$ in all $Q_j$, $1 \leq j \leq k$, are, in fact, all the
indices from 1 through \( n'_k \). Substituting \( m \) for \( k \) here yields the preceding assertion, since \( n'_m = n \). Our assertion holds vacuously for \( k = 0 \); assuming it for \( k - 1 \), we show it for \( k \). The range from 1 through \( n'_k \) is the range from 1 through \( n'_{k-1} \) followed disjointly by the range from \( n'_{k-1} + 1 \) through \( n_k = n'_k \). The new \( x(X, T) \) added in the larger range are those for \( X \) in \( Q_k \); since \( c_X = k \), for each \( X \) in that range, we have \( x(X, T) = n'_{k-1} + x(X, T') \). By inductive assumption, all the \( x(X, T') \) are in the range from 1 through \( n_k \), from which the proof for \( k \) follows.

We now proceed to part (a) above. Let \( E = U \to V \) be an edge in \( G \), and suppose that \( U = P_i \) is in \( Q_a \) and \( V = P_j \) is in \( Q_b \). If \( a = b \), then \( U \) and \( V \) are both in \( Q_a \), having loop tree \( T' \). By inductive assumption, if \( E = P_i \to P_j \) in \( Q_a \), we have \( i' < j' \) unless \( E \) is a loop in \( T' \). However, in this case, \( c_U = a = b = c_V \); letting \( z = n'_{cU-1} \), we have \( i' = x(U, T') < x(V, T') = j' \) if and only if \( i = x(U, T) = z + x(U, T') < z + x(V, T') = x(V, T) = j \).

Also, by Definition 9, all loopbacks in \( T' \) are also loopbacks in \( T \).

If \( a \neq b \), then, by Definition 5, there is an edge in \( G' \) from \( Q_a \) to \( Q_b \); since \( Q_1, \ldots, Q_m \) is a topological sort of \( G' \), we therefore, have \( a < b \) by Definition 6. By Eq. (1), we have

\[
\begin{align*}
    i &= x(U, T) \\
    &= n'_{i-1} + x(U, T') \quad \text{(since } x(U, T') \leq n_a) \\
    &= n'_{i-1} + n_a \\
    &< n'_a + x(V, T') \quad \text{(since } x(V, T') > 0) \\
    &\leq n'_{b-1} + x(V, T') \quad \text{(since } a < b, \text{ so that } a \leq b - 1) \\
    &= x(V, T) \\
    &= j
\end{align*}
\]

so that \( i < j \), completing the proof of part (a).

Finally, we show part (b) above. Let \( L \) be a loop of level \( n \) in \( T \); we proceed by induction on \( n \). If \( n = 1 \), then, using the notation above, \( L \) is some \( Q_k \); by Eq. (1), the values of \( x(X, T) \), for \( X \) in \( L \), range from \( n'_{k-1} + 1 \) through \( n'_{k} + n \). It remains to show that \( x(H, T) = n'_{k-1} + 1 \), where \( H \) is the head of \( L \). Here \( H \), by itself, is a trivial strong component of the body \( B \) of \( Q_k \), by Lemma 9, and is therefore in the component graph \( B' \) of \( B \). Also, \( H \) is the start node of \( B \), as a rooted graph, by Lemma 10; and therefore \( H \), by itself, is the start node of \( B' \), by Definition 5. Within \( B \), \( H \) has indegree 0, since all edges leading to \( H \) are loopbacks and were removed from \( L \) to form \( B \), by Definition 9(a). It therefore follows from Definition 5 that \( H \), by itself, has indegree 0 in \( B' \). By Lemma 5, therefore, \( H \) may be taken as \( P_1 \) in a topological sort of \( B' \). Let \( T' \) be the subtree of \( T \) with root \( L \), and let \( T'' \) be the subtree of \( T' \) with root \( P_1 \). In using Eq. (1):

- to calculate \( x(H, T'') \), we have \( n_1 = 1 \), and \( T'' \) has height 0, so that \( x(H, T'') = 1 \).
- to calculate \( x(H, T') \), we have \( c_H = 1 \) (since \( H \) is in \( P_1 \)), so that \( n'_{c_H-1} = n'_0 = 0 \), and \( x(H, T') = 0 + x(H, T'') = 1 \).
- to calculate \( x(H, T) \), we have \( x(H, T) = n'_{k-1} + x(H, T') = n'_{k-1} + 1 \).

If \( n > 1 \), then, by Definition 9(b), \( L \) is a loop at level \( n - 1 \) in the body \( B \) of some non-trivial strong component of an outer loop \( L' \) of \( G \). Let \( T' \) be the subtree of \( T \) with root \( L' \). By inductive hypothesis, there is an ordering \( P'_1, \ldots, P'_n \) of the nodes of \( B \), such that, if \( L \) has \( k \) nodes, and head \( P'_1 \), the nodes of \( L \) are precisely \( P'_1, P'_{i+1}, \ldots, P'_{i+k-1} \). Here \( x(X, T') \), where \( X \) is a node in \( B \), is the index of \( X \) in the ordering \( P'_1, \ldots, P'_n \) (that is, \( x(P'_j, T) = j \)); and this is the \( x(X, T') \) that appears in Eq. (1) above. If \( L' \) is some \( Q_k \) in the ordering of all outer loops of \( G \) in its component graph \( G' \), then, for every node \( X \) in \( L \), we have \( c_X = k \). In particular, the values of \( c_X \), for all these \( X \), are the same; and we denote \( k - 1 \) by \( j \). If now the various \( x(X, T') \), for \( X \) in \( L \), range from \( P'_1 \) through \( P'_{i+k-1} \), then, by Eq. (1), the various \( x(X, T) \), for these same values of \( X \), range from \( n'_{j} + P'_1 \) through \( n'_{j} + P'_{i+k-1} \). If \( H \) is the head of \( L \), then \( x(H, T') = P'_i \), by inductive hypothesis; and so \( x(H, T) = n'_{j} + P'_i \), completing the proof of part (b).

8. Reordering unstructured programs

Theorem 1 may be used in reordering an existing unstructured program, as illustrated by the following four examples.
8.1. A depth-first search program

We first solve the problem which we posed in Fig. 4, in Section 2 above. The solution is given in Fig. 8; this solution was obtained as follows:

* First we need to determine the non-trivial strong components of the graph of Fig. 4. Several methods of doing this constructively are known (see Section 9.2 below). In this case the strong components are L1, induced by \{P2, P3, P4, P6, P10, P11\}, and L2, induced by \{P7, P8\}. It is easy to verify, from Fig. 4, that L1 and L2 are both strongly connected.

* Looking first at L1, we see that it has exactly one entry point, namely P3 (there is an edge to P3 from P1, which is outside L1). Therefore P3 is the head of L1.

* The only loopback in L1 is the edge from P2 (which is in L1) to P3 (the head of L1). The body B1 of L1 is L1 with this loopback removed.

* We now need to find non-trivial strong components of B1, just as we did before. There is only one of these, namely L1A, induced by \{P2, P5, P6, P10, P11\}. As before, it is easy to verify that L1A is strongly connected.

* Here L1A has exactly one entry point, namely P5 (there is an edge to P5 from P4, which is outside L1A). Therefore P5 is the head of L1A.

* The only loopback in L1A is the edge from P11 (which is in L1A) to P5 (the head of L1A). The body B1A of L1A is L1A with this loopback removed. It is not hard to see that B1A is acyclic; so L1A has no inner loops.

* Looking now at L2, we see that it has exactly one entry point, namely P7 (there is an edge to P7 from P6, which is outside L1). Therefore P7 is the head of L1.

* The only loopback in L2 is the edge from P8 (which is in L2) to P7 (the head of L2). The body B2 of L2 is L2 with this loopback removed. It is not hard to see that B2 is acyclic; so L2 has no inner loops.

* The forward edge from P6 to P12 shows why this program is generalized structured, not well-structured (see Section 1 above).

8.2. Syntactic and semantic loop trees

In the example of Fig. 8, there were no syntactic loop trees. We have seen that syntactic and semantic loop trees are not always the same, but there are cases, as in Fig. 9, where they are close to being the same. This figure shows the construction of the loop tree of a graph like that of Fig. 3, with the loops E, F, and G relabeled L1, L2, and L3, respectively. Starting with Fig. 9(a), these loops are constructed; the final result, with all the loopbacks removed, is given in Fig. 9(j). In this example, each node in the loop tree has no more than one child, and every loop has exactly one loopback. Also, this example is reducible, in the following sense.

**Definition 10.** A graph is reducible if it has exactly one loop tree \(T\), with every loop in \(T\) having a unique head.

In [29], we have shown that reducibility, in this sense, is equivalent to reducibility in the sense of Allen and Cocke [5].
The flow graph (a), like that of Fig. 3, has one non-trivial strong component, $L1$ (b). The only entry point of $L1$ is 2 (c); the only loopback is $15 \to 2$, which we remove (d), leaving a graph with, again, one non-trivial strong component, $L2$ (e). The only entry point of $L2$ is 4 (f); the only loopback is $13 \to 4$, which we remove (g), leaving, once more, a graph with one non-trivial strong component, $L3$ (h). The only entry point of $L3$ is 7 (i); the only loopback is $10 \to 7$, which we remove (j), leaving a dag.

Fig. 9. Explaining the semantic loop tree of a graph like that of Fig. 3.

8.3. Multiple loop trees

The semantic loop trees of the graphs of Fig. 4 and of Fig. 9 were unique; but a graph can also have more than one semantic loop tree, as illustrated in Fig. 10. As we start constructing a loop tree for $G$, we see that it has two strong components, $LA$ and $LB$. Therefore, these become children of $G$ in the loop tree, as we see in Fig. 10(c). This time, however, nodes 5 and 6 are both possible entry points for $LB$; and so there are two possible loop trees, of different
A graph $G$ (a) with two non-trivial strong components, $LA$ and $LB$ (b), which are both children of $G$ in the loop tree (c). The only entry point of $LA$ is 2 (d), and the only loopback is $4 \rightarrow 2$, which we remove (e), leaving a dag. If we choose 5 as an entry point for $LB$ (f), there is now one loopback, $8 \rightarrow 5$, which we remove (g), leaving an inner loop $LC$ (h) in the loop tree (i). The only entry point of $LC$ is 6 (j), and the only loopback is $8 \rightarrow 6$, which we remove (k), leaving a dag. If we choose 6 as an entry point for $LB$ (l), there are now two loopbacks, $5 \rightarrow 6$ and $8 \rightarrow 6$; and removing these (m) leaves a dag.

Fig. 10. A graph with two different semantic loop trees.

forms. If node 5 is the head of $LB$, then $LB$ has an inner loop, $LC$, which is a child of $LB$ in the loop tree, as we see in Fig. 10(i). If node 6 is the head of $LB$, then $LB$ has two loopbacks, $5 \rightarrow 6$ and $8 \rightarrow 6$, but no inner loop; so the loop tree, in this case, remains that of Fig. 10(c).

The numbering of the nodes of a graph, as specified by Theorem 1, is given in the example of Fig. 10, using 5 as the entry point for $LB$. Let $BA$ be the body of $LA$ (with head 2); let $BB$ be the body of $LB$ (with head 5); and let $BC$
be the body of \( LC \) (with head 6). Let \( TA, TB, \) and \( TC \) be the subtrees of \( G \), here, which have the respective roots \( LA, LB, \) and \( LC \). Then:

- The only topological sort of \( BC \) (see Fig. 10(k)) is in the order (6, 7, 8). Thus \( x(6, TC) = 1; x(7, TC) = 2; \) and \( x(8, TC) = 3. \)
- The component graph of \( BB \) contains its strong components, \{5\} and \( LC \). The only topological sort of this is in the order \((\{5\}, LC)\). At this level, \( n_1 = 1 \) (there is one node in \{5\}) and \( n_2 = 3 \) (there are three nodes in \( LC \), so that \( n'_1 \) is also 1 (and \( n'_0 = 0 \)). We thus have:
  - \( x(5, TB) = n'_0 + x(5, \{5\}) = 0 + 1 = 1 \)
  - \( x(6, TB) = n'_1 + x(6, TC) = 1 + 1 = 2 \)
  - \( x(7, TB) = n'_1 + x(7, TC) = 1 + 2 = 3 \)
  - \( x(8, TB) = n'_1 + x(8, TC) = 1 + 3 = 4. \)
- The only topological sort of \( BA \) (see Fig. 10(e)) is in the order (2, 3, 4). Thus \( x(2, TA) = 1; x(3, TA) = 2; \) and \( x(4, TA) = 3. \)
- The component graph of \( G \) contains its strong components, \{1\}, \( LA, \) and \( LB \). The only topological sort of this is in the order \((\{1\}, LA, LB)\). At this level, \( n_1 = 1 \) (there is one node in \{1\}); \( n_2 = 3 \) (there are three nodes in \( LA \)); and \( n_3 = 4 \) (there are four nodes in \( LB \)). Therefore \( n'_1 = 1; n'_2 = 1 + 3 = 4; \) and \( n'_0 = 0. \) We thus have:
  - \( x(1, G) = n'_0 + x(1, \{1\}) = 0 + 1 = 1 \)
  - \( x(2, G) = n'_1 + x(2, TA) = 1 + 1 = 2 \)
  - \( x(3, G) = n'_1 + x(3, TA) = 1 + 2 = 3 \)
  - \( x(4, G) = n'_1 + x(4, TA) = 1 + 3 = 4 \)
  - \( x(5, G) = n'_2 + x(5, TB) = 4 + 1 = 5 \)
  - \( x(6, G) = n'_2 + x(6, TB) = 4 + 2 = 6 \)
  - \( x(7, G) = n'_2 + x(7, TB) = 4 + 3 = 7 \)
  - \( x(8, G) = n'_2 + x(8, TB) = 4 + 4 = 8. \)
- Thus we see that \( x(i, G) = i \) for \( 1 \leq i \leq 8. \) (This is in fact, why we numbered the nodes in this way, in Fig. 10.)

### 8.4. Compiled programs

Rearrangement may sometimes be applied to programs written in higher-level languages (see Section 9.4 below). An example of such a rearranged program is given in Fig. 11. This program was produced by the author as a result of a series of transformations applied to the heap sort program known as Treessort [14]. The result is considerably shorter, and free from the procedure call overhead of the program of [14]; but it contains one unconditional and five conditional \texttt{goto} statements, and gives no clue as to how it could be rearranged. Using Theorem 1, we reordered it to produce the GSP of Fig. 11(b). In the process, statement 6 in the original program has disappeared, since it indicates, there, that statement 5 is immediately followed by statement 9. (In the new program, statement 9 follows statement 5 directly.)

Once may verify immediately that the meaning of the program of Fig. 11(b) is exactly, statement for statement, that of the program of Fig. 11(a). As an example, statement 13, in Fig. 11(a), is \( \text{if} (t[m] > k) \text{goto} l7; \) which goes to statement 7 if \( t[m] > k \) and proceeds to statement 14 (the next statement) if \( t[m] \leq k. \) In Fig. 11(b), statement 13 is if \( (t[m] \leq k) \text{break}; \) which proceeds to statement 7 (the next statement) if \( t[m] > k \) and breaks out of the innermost \texttt{while} loop if \( t[m] \leq k. \) The statement immediately following that loop is statement 14; so the meaning of the given statement is the same in either case. Similar considerations apply to all the other statements of this program.

Note that the program of Fig. 11(a) is difficult to understand, but not just because it is unstructured. Indeed, the program of Fig. 11(b) is just as difficult to understand, even though it now has a structure. This illustrates why it would be unwise to use Theorem 1 to justify the construction of new unstructured programs, hoping to rearrange them later on. Nevertheless, there are currently a wide variety of old unstructured programs waiting to be rearranged. As we have just seen, the example of Fig. 11 shows that rearrangement is not the complete answer to the problem of understandability; but without rearrangement into generalized structured form, it is easy to argue that we cannot even start trying to understand an unstructured program. This can pose security concerns, if the unstructured program is vital to the operations of some organization.
and loops loops component of $G$ loops as before. The nodes of $M$ necessary in representing a tree). This may be compared to the representation of a min-heap or a max-heap

9. Implementations

We now take up the use of the computer in determining loop trees and rearranging programs.

9.1. A loop tree data structure

Theorem 1(b) leads to a compact representation of a loop tree, without using any pointers (as would normally be necessary in representing a tree). This may be compared to the representation of a min-heap or a max-heap $Z$ for the purposes of heap sorting; even though $Z$ is a tree, it is still represented as an array, without using pointers. In a similar way, a loop tree $T$ of a graph $G$ having $n$ nodes may be represented by two arrays of length $n$, which we call order and loops. If the nodes of $G$ are taken to be integers from 0 through $n-1$, then order[0] through order[$n-1$] are these integers in the new order.

The loops array specifies where the loops are. If the node $N$ with index order[$k$], by itself, is a trivial strong component of $G$, then loops[$k$] = $k$. Otherwise, $N$ is contained in some outer loop in $G$. If it is the head of an outer loop $L$ containing $j$ nodes, then loops[$k$] = $k + j$. This includes the case in which $L$ is a self-loop, so that $j$ is 1 and loops[$k$] = $k + 1$. By Theorem 1(b), the nodes of $L$ are now order[$k$], order[$k + 1$], ..., order[$k + j - 1$].

Now suppose that $N$ is in $L$, but is not the head of $L$. If $N$ is a trivial strong component of the body of $L$, we again have loops[$k$] = $k$. If it is the head of an inner loop $M$ containing $j$ nodes, then loops[$k$] = $k + j$, where $j$ might be 1, as before. The nodes of $M$ are now order[$k$], order[$k + 1$], ..., order[$k + j - 1$], again as before. These specifications are further repeated if $N$ is in a loop at level greater than 2. Note that every node in the graph is either a trivial strong component at some level, or a loop head at some level.

Fig. 11. Rearrangement of an unstructured heap-sorting program.
9.2. Loop tree builders

In order to set up the order and loops arrays for a graph $G$, as indicated above, we need to determine its strong components. Ordinary components can be found by a simple depth-first search; strong components are a little harder. However, there exist several strong component algorithms, beginning with the work of Tarjan [37]. Perhaps the simplest of these is due to Sharir [36], based on unpublished work of Kosaraju and appearing in standard texts on algorithms, such as [2] and [11]. This algorithm requires precomputation of the transpose of $G$; a more recent algorithm, which avoids this precomputation, has been found by Gabow [15]. Using either of these algorithms, a loop tree can be found in time no worse than $O(n^2)$, where the worst case is that of a complete graph having $n$ nodes and $n^2$ edges. Here every loop tree has height $n$, and the $k$th loop in a typical loop tree consists of nodes 1 through $k$.

The algorithm of [36] does two depth-first searches, one on the original graph and one on its transpose. The order of start vertices in the second DFS is the reverse of the order of finished vertices in the first DFS. Using this algorithm, we have constructed two loop tree builders. In one, the vertices are represented as small integers; in the other, they are represented as strings, not containing whitespace. In either case, an edge is represented by a pair of vertices, and the loop tree builder provides a visual picture of one of the resulting loop trees.

9.3. Assembly language rearrangers

The design of a loop tree builder may be extended to that of a program which rearranges code written in some programming language $L$. We refer to such a programming utility as a rearranger for the language $L$. Such a program takes, as input, a program $P$ written in $L$; constructs the flow graph $G$ of $P$, using the syntactic rules of $L$; uses the loop tree builder to construct a loop tree for $G$, with the order and loops arrays as described in Section 9.1 above; and then uses these arrays to construct a rearrangement of $P$ into a GSP.

We have constructed a rearranger for assembly-language programs, and reported upon it in [28]. It inputs a description $D$ of an assembler $A$, followed by a program $P$ written in $A$, and rearranges $P$. Many ways in which assemblers differ among themselves are embodied in $D$; however, since this is a “universal” rearranger, it places severe restrictions on $P$ and $A$. Nevertheless, we have been able to construct descriptions of subsets of five assembly languages (mainframe, Intel, 68000, G3, and MIPS); translate an unstructured depth-first search (used by the rearranger itself) into each of these languages; and rearrange it for each of these languages.

In Section 1 above, we mentioned that there may be minor syntactic differences between an assembly language program and its rearrangement. These may include elimination of certain unconditional goto statements; the introduction of new ones; the replacement of certain conditional goto statements by their opposites (replacing Branch on Less Than by Branch on Greater Than or Equal, for example); and giving labels to certain previously unlabeled statements. Specifics of this are taken up further in [28].

9.4. Further rearrangers

The amount of work required to produce rearrangers for the variety of existing programming languages is substantial, and we regard the construction of such rearrangers as central to our projected further work in this field.

Let us consider machine language (that is, binary instruction codes) first. It is possible that a machine language program would have to be rearranged, because it was written first in assembler or in COBOL and was later patched at the machine level (this was common in the 1960s). It would seem, however, that there is no need to write a separate rearranger for this case. We should be able to disassemble the machine language program, rearrange the resulting assembly language code, and then assemble the result back again.

Compiled code presents further problems. In Section 8.4 above, we gave an example of a rearranged compiled program; but that program was written in C, which allows the goto statement. In general, the result of rearrangement will be a GSP, and therefore might contain forward goto statements, which are disallowed in certain programming languages. Programs written in such a language cannot necessarily be rearranged into other programs written in that same language. This subject is taken up further in Section 11.2 below.
10. Scalability

Any algorithm based on graph search raises scalability concerns, because searching a graph requires the entire graph to be in memory at once. This is in contrast to an assembler, for example, in which only the symbol table has to be all in memory at once. The source program does not, because that is read sequentially, from beginning to end, either once or twice (for one-pass or two-pass assemblers, respectively). An assembler can therefore process a large-scale program as easily as a small-scale one, provided that the symbol table does not get too large. For a general graph search algorithm, this is not the case.

We first note that, by a large-scale program, we do not mean a program together with all its subprograms; a rearranger acts on one subprogram at a time. In particular, when a statement \( U \) in a program \( F \) calls a subprogram \( G \), there is no edge in any graph, processed by the rearranger, from \( U \) to the start of \( G \). The graph of \( F \) and the graph of \( G \) are separate, and a single edge in the graph of \( F \) leads from the statement \( U \) to the statement \( U' \) that follows \( U \).

Thus the entire subprogram \( G \) is thought of, within \( F \), as if it were a single statement of \( F \).

This treatment of \( F \) must be slightly modified if \( G \) does not necessarily return to \( U' \). If \( G \) returns to some statement \( U'' \) within \( F \), then two edges lead from \( U \) in the graph of \( F \), one to \( U' \) and one to \( U'' \). If \( U'' \), here, is in some subprogram that calls \( F \), then again two edges lead from \( U \), one to \( U' \) and one to a statement which exits from \( F \). Neither of these cases change the fact that \( F \) and \( G \) are taken to have separate graphs. However, a single program, often the main program, might still be too large to be rearranged, under certain conditions.

We now show how to modify a rearranger, as in Section 9.3 above, so that it will have roughly the same scalability that an assembler does. This modification is based on the fact that directed graphs which can be processed by a rearranger are far more general than real flow graphs. A real flow graph arises from a real program, in which every statement always proceeds either to the next statement or to some labeled statement. This may be formalized as follows.

**Definition 11.** A labeled program is a program having a flow graph, rooted at some node \( S \), whose nodes can be ordered as \( P_1, \ldots, P_n \), such that:

(a) Some of the nodes, including \( P_1 \) and \( S \) (possibly \( S = P_1 \)), are identified as labeled statements.

(b) For each \( i, 1 \leq i \leq n - 1 \), every edge from \( P_i \) must go either to \( P_{i+1} \) or to some labeled statement.

Most assembly language and machine language programs are labeled in a natural way; whereas most compiled programs can easily be labeled by attaching generated labels to certain statements (while, do-while, for, case [in the C sense], and else) and to the statements following them.

In Definition 13 below, we define, for each labeled program, a label graph \( G' \), whose loop trees correspond to loop trees of the flow graph, as shown in Theorem 3 below. The nodes of \( G' \) are the labels of executable statements in the program, so \( G' \) is normally much smaller than the flow graph \( G \). Indeed, the size of \( G' \), in memory, is comparable to the size of the symbol table of an assembler, since they are both based on labels. Here \( G' \) does not contain the data labels which must be in an assembler symbol table, although \( G' \) also contains adjacency lists, which increases its size somewhat.

**Definition 12.** Given the label of an executable statement \( X \) in a labeled program \( P \), the label interval \( L I(X) \) consists of \( X \) together with all statements which follow \( X \) in \( P \), up to, but not including, the next labeled statement after \( X \) (or to the end of \( P \), if no labeled statement in \( P \) follows \( X \)).

Since \( P_1 \) is labeled, it follows that every statement in \( P \) is in some label interval of \( P \). Also, the label intervals may be found by going through \( P \) sequentially, from beginning to end, thus promoting scalability. This is in contrast to other kinds of intervals. For example, let \( Q_1, Q_2, \ldots, Q_n \) be arbitrary assignment statements in a program, with each \( Q_i \) being followed by \( J_i \), which is an unconditional jump to \( Q_{i+1} \), for \( 1 \leq i \leq n - 1 \). The sequence \( Q_1, J_1, \ldots, Q_{n-1}, J_{n-1}, Q_n \) acts, semantically, like an interval; but, if the \( Q_i \) are scattered all over the program, such an interval cannot be found scalably.

**Lemma 11.** If the statements in \( L I(X) \) are \( X_1, X_2, \ldots, X_n \), in that order, then there is exactly one edge in the graph to \( X_i \) for all \( i, 2 \leq i \leq n \), namely from \( X_{i-1} \).

**Proof.** If there is no edge to \( X_i \) at all, then \( X_i \) is not reachable from the start node \( S \) (note that \( X_i \) is not \( S \), since \( S \) is labeled, by Definition 11, while \( X_i \) is not). This contradicts our assumption, in Definition 11, that \( P \) is rooted at \( S \).
(That is the reason for this restriction on \( P \).) But \( X_i \) can have an edge to it only from \( X_{i-1} \), since all edges other than this edge are to labeled statements, by Definition 11(b), and \( X_i \) is not labeled, for \( 2 \leq i \leq n \), by Definition 12. \qed

**Definition 13.** The **label graph** of a labeled program \( P \), with flow graph \( G \), is a graph \( G' \) whose nodes are the labeled statements of \( P \), with an edge from \( X \) to \( X' \) in \( G' \) denoting a path from \( X \) to \( X' \) in \( G \), entirely within \( LI(X) \) except possibly for its last statement. (There may be more than one such path, but multiple edges in the label graph are not allowed. Self-loops, however, are allowed, if \( X = X' \) in the above.) The start node of \( G' \) is taken to be the start node of \( G \) (which is labeled, by Definition 11).

**Definition 14.** Let \( \pi' = LI(X_1) \rightarrow LI(X_2) \rightarrow \cdots \rightarrow LI(X_n) \) be a path in the label graph \( G' \) of \( P \). For each \( i \), \( 1 \leq i \leq n - 1 \), let \( \pi_i \) be a path from \( X_i \) to \( X_{i+1} \) in the flow graph \( G \) of \( P \), as given by Definition 13 above. Then the concatenation \( \pi \) of \( \pi_1 \) through \( \pi_{n-1} \) is an expanded path in \( G \) corresponding to the path \( \pi' \) in \( G' \). (In general, \( \pi' \) might have more than one expanded path.) An expanded cycle is an expanded path of a cycle in the label graph.

It follows immediately from Definitions 13 and 14 that every path \( \pi' \) in \( G' \) has at least one expanded path \( \pi \) in \( G \), and that \( \pi \) contains all the nodes of \( \pi' \). Every expanded path must go from one labeled statement to another; so not every path in a flow graph is an expanded path. However, we have the following.

**Lemma 12.** Every cycle \( C \) in the flow graph of a labeled program \( P \) contains at least one labeled statement; and \( C \) is an expanded cycle of some cycle \( C' \) in the label graph \( G' \) of \( P \).

**Proof.** Let \( j \) be the smallest integer such that \( P_j \) is in \( C \). Since \( C \) is a cycle, there must be some edge in \( C \) leading to \( P_j \). Since \( P_{j-1} \) is not in \( C \), by assumption, that edge cannot lead from \( P_j \). Therefore, by Definition 11(b), \( P_j \) must be labeled. For the second part of the lemma, let \( P_j = X_1, X_2, \ldots, X_k \) be all the labeled statements of \( C \), in order. For each \( i \), \( 1 \leq i \leq k - 1 \), the statements in \( C \) between \( X_i \) and \( X_{i+1} \) must all be in \( LI(X_i) \); they must be in the original order up to some statement which, by Definition 11, is labeled and therefore must be \( X_{i+1} \). Therefore \( P_j = X_1 \rightarrow X_2 \rightarrow \cdots \rightarrow X_k = P_j \) is a cycle \( C' \) in \( G' \), by Definition 13, and \( C \) is an expanded cycle of \( C' \), by Definition 14. \qed

**Lemma 13.** Let \( X_1 (=X), X_2, \ldots, X_n \) be the statements in \( LI(X) \), and let \( Y = X_k \). Any cycle \( C \) which contains \( Y \) must contain all \( X_i \), for \( 1 \leq i \leq k \); in particular, \( C \) must contain \( X \).

**Proof.** Suppose the contrary, so that there are some values of \( i < k \) for which \( X_i \) is not in \( C \). Let \( j \) be the largest of these values of \( i \), so that \( X_j \) is not in \( C \), but \( X_{j+1} \) is in \( C \). Clearly \( X_{j+1} \) exists, since \( X_k \) is in \( C \). However, the only edge to \( X_{j+1} \) is from \( X_j \), by Lemma 11, a contradiction. \qed

**Lemma 14.** Let \( M \) be the strong component of a graph \( G \) which contains a node \( N \) in \( G \). Suppose that \( N \) is contained in a strongly connected subset \( M' \) of the label graph \( G' \) of \( G \). Then \( M' \) is contained in \( M \).

**Proof.** Let \( V \) be in \( M' \). Since \( M' \) is strongly connected, there are paths from \( N \) to \( V \) and from \( V \) to \( N \) in \( G' \). These paths have expanded paths in \( G \), by Definition 14, from \( N \) to \( V \) and from \( V \) to \( N \). Therefore \( V \) is in \( M \) by Definition 3. \qed

**Theorem 2.** Let \( G \) be the flow graph of a labeled program \( P \), and let \( G' \) be its label graph, as in Definition 13. Let \( L \) be a non-trivial strong component of \( G \), and let \( L' \) be induced, in \( G' \), by the set of all labeled statements in \( L \). Then \( L' \) is a non-trivial strong component of \( G' \). Conversely, let \( L' \) be a non-trivial strong component of \( G' \) and let \( Y \) be the union of all \( LI(J) \), for \( J \) in \( L' \). Then \( Y \) is the (disjoint) union of a non-trivial strong component \( M \) of \( G \) with (zero or more) unlabeled trivial strong components of \( G \); and \( L' \) is the set of all labeled statements in \( M \).

**Proof.** Since \( L \) is non-trivial, it contains at least one edge, and therefore at least one cycle. By Lemma 12, this cycle contains a labeled statement \( N \). Let \( L'' \) be the strong component of \( G' \) which contains \( N \); we show that \( L'' = L' \). If \( K \) is in \( L'' \) (and therefore labeled), there is a cycle \( C' \) in \( L'' \) containing both \( N \) and \( K \). Let \( C \) be an expanded cycle of \( C' \) in \( G \), as in Definition 14; then \( C \) contains all nodes of \( C' \). In particular, \( C \) contains both \( N \) and \( K \), and therefore \( K \) is in \( L \), by Lemma 7; since \( K \) is labeled, it is also in \( L' \). Conversely, let \( K \) be in \( L' \) (so that, again, \( K \) is labeled). Then \( K \) is in \( L \), and, by Lemma 7, there is a cycle (call it \( C_K \) in \( G \), of non-zero length, which contains both \( N \) and \( K \). Again by Lemma 12, \( C_K \) is an expanded cycle of some cycle (call it \( D_K \) in \( G' \)). It follows immediately from Definition 14 that \( D_K \) also contains both \( N \) and \( K \), since \( K \) is labeled. But then, by Lemma 7, \( K \) is in \( L'' \).
We now pass to the second part of the theorem. Since \( L' \) is non-trivial, it contains at least one edge, and therefore at least one cycle. Let \( N \) be in that cycle, and let \( M \) be the strong component of \( G \) that contains all nodes in \( L' \); in particular, \( N \) is in \( M \). We show that \( M \) is contained in \( Y \). Suppose the contrary, so that there is a node \( K \) in \( M \) that is not in \( Y \). Here \( K \) is in \( LI(J) \) for some labeled statement \( J \); and \( J \) is not in \( L' \), because otherwise \( K \), being in \( LI(J) \), would be in \( Y \), by the definition of \( Y \). Since the strong component \( M \) contains both \( N \) and \( K \), it contains a cycle \( C \) that contains both \( N \) and \( K \). By Lemma 13, \( C \) also contains \( J \). By Lemma 12, \( C \) is an expanded cycle of some cycle \( C' \) in \( G' \), which also contains both \( N \) and \( J \). The union of \( C' \) and \( L' \) is strongly connected by Lemma 2, since \( N \) is in both \( C' \) and \( L' \); and this contradicts Lemma 1, since \( J \) is not in \( L' \).

Now let \( K \) be in \( Y \), but not in \( M \), so that \( K \) is in \( LI(J) \), for some \( J \) in \( L' \), by definition. We show that \( K \), by itself, is a trivial strong component of \( G \). Suppose the contrary; then \( K \) is in some cycle \( C \) in \( G \). By Lemma 13, \( C \) must contain \( J \), which is labeled. Also, \( M \), by definition, contains all nodes in \( L' \), and thus contains \( J \). Therefore \( C \), which contains \( J \), is contained in \( M \) by Lemma 3. But then \( K \), which is in \( C \), is also in \( M \), contrary to hypothesis.

We now show that \( K \) is unlabeled. Suppose the contrary; then, since \( K \) is in \( LI(J) \), we must have \( K = J \), since \( J \) is the only labeled statement in \( LI(J) \), by Definition 12. Therefore, \( K \) is in \( L' \). But \( L' \) is a non-trivial strong component of \( G' \), implying that \( K \) is in a cycle in \( L' \). This is a contradiction, since \( K \), by itself, is a trivial strong component of \( G \).

Finally we show that \( L' \) is in fact the set of all labeled statements in \( M \). Since \( M \) is contained in \( Y \), any labeled statement \( K \) in \( M \) is also in \( Y \). By the definition of \( Y \), we see that \( K \) is in some \( LI(J) \), for \( J \) in \( L' \). However, as before, \( J \) is the only labeled statement in \( LI(J) \), so that \( K = J \) and \( K \) is in \( L' \). Conversely, if \( K \) is in \( L' \), then \( K \) is labeled, by the definition of \( L' \); and \( LI(K) \) is in \( Y \), by the definition of \( Y \). Thus \( K \) is in \( Y \), and, in fact, \( K \) is in \( M \), since otherwise \( K \) would be unlabeled.

Theorem 2 gives rise to the following definition.

**Definition 15.** Given an outer loop \( L \) in the flow graph \( G \) of a labeled program \( P \) (that is, a non-trivial strong component of \( G \)), the **label loop** \( L' \) in the label graph \( G' \) of \( P \) is induced by the set of labeled statements in \( L \); this is an outer loop of \( G' \) by Theorem 2. More generally, if \( L \) is a loop of level \( n > 1 \), anywhere within a loop tree \( T \) of \( P \), its **label loop** \( L' \) in the label graph \( G' \) of \( P \) is induced by the set of labeled statements in \( L \).

Given a loop tree for \( G \), there is a corresponding loop tree for \( G' \) (see Definition 16 below). To set this up, we need some further lemmas.

**Lemma 15.** An outer loop \( L \) containing any node \( N \) in a label interval \( LI(X) \) must contain all nodes (including \( X \)) which precede \( N \) in \( LI(X) \).

**Proof.** Since \( L \) is strongly connected, it contains some cycle \( C \) which contains \( N \). However, by Lemma 13, \( C \), and therefore \( L \), must contain all nodes which precede \( N \) in \( LI(X) \).

**Lemma 16.** The head \( H \) of a loop \( L \) must be a labeled statement.

**Proof.** If \( H \) is the start statement of the graph, it is labeled, by Definition 11. Otherwise, \( H \) is an entry point of \( L \), by Definition 8. Suppose that \( H \) is in some \( LI(X) \) and is not \( X \). There must be an edge to \( H \) from outside \( L \); but the only edge to \( H \) is from the statement \( K \) preceding \( H \) in \( LI(X) \), by Lemma 11. However, by Lemma 15, \( L \) contains \( K \), which contradicts our assumption that \( K \) is outside \( L \).

**Lemma 17.** Using the terminology of Definition 15, \( H \) can be the head of an outer loop \( L \) in \( G \) if and only if it can be a head of the label loop \( L' \) of \( L \) in \( G' \).

**Proof.** If \( H \) is the start node of \( G \), it is also the start node of \( G' \), by Definition 13, and we are done. Now let \( H \) be an entry point of \( L \), so there is an edge to \( H \) from some node \( K \) in \( G \) outside \( L \). By Lemma 16, \( H \) is labeled, and is therefore in \( G' \), by Definition 13. Let \( K \) be in \( LI(X) \); there is then, by Lemma 11, a path in \( G \) (possibly of length zero) from \( X \) to \( K \). Following this by the edge from \( K \) to \( H \), we obtain a path from \( X \) to \( H \). By Definition 13, there is then an edge from \( X \) to \( H \) in \( G' \); we now show that \( H \) is also an entry point of \( L' \) by showing that \( X \) is not in \( L' \). Suppose the contrary, so that there is a cycle \( C' \) in \( L' \) containing both \( X \) and \( H \). Consider any expanded cycle \( C \) of \( C' \); this also contains both \( X \) and \( H \), by Definition 14. If \( C \) does not include \( K \), we replace the portion of \( C \) between \( X \) and \( H \) by the path, constructed above, from \( X \) through \( K \) to \( H \). The resulting cycle (call it \( C'' \)) contains \( H \) and is
Thus contained in the strong component \( L \), by Lemma 3. However, \( C'' \) also contains \( K \), so that \( K \) is in \( L \), contrary to hypothesis.

Conversely, let \( H \) be an entry point of \( L' \), so that there is an edge in \( G' \) to \( H \) from some node \( K \) in \( G' \) outside \( L' \). Since \( K \) is in \( G' \), it is labeled. If \( K \) were in \( L \), then, since \( K \) is labeled, it would also be in \( L' \); a contradiction; hence \( K \) is outside \( L \). Since there is an edge from \( K \) to \( H \) in \( G' \), there is a path from \( K \) through \( LI(K) \) to \( H \) in \( G' \), by Definition 13. Let \( K' \) be the last node on this path before \( H \), so that there is an edge in \( G \) from \( K' \) to \( H \). We show that \( H \) is an entry point of \( L' \) by showing that \( K' \) is outside \( L \). Suppose the contrary; then \( K' \) is in a cycle \( C \) in \( L \). By Lemma 13, \( C \) must contain \( K \), since \( K' \) is in \( LI(K) \). The union of \( C \) and \( L \) is strongly connected by Lemma 2, since \( K' \) is in both \( C \) and \( L \); and this union properly contains \( L \), since \( K \) is outside \( L \). This, however, contradicts Lemma 1, since \( L \) is a strong component of \( G \). □

**Lemma 18.** The label loop \( B' \) of the body \( B \) of a loop \( L \) with head \( H \) is the body of the label loop \( L' \) of \( L \) with head \( H \) (which is a head of \( L' \) by Lemma 17).

**Proof.** Let \( K \rightarrow H \) be any loopback in \( L \), and let \( K \) be in \( LI(X) \). By Lemma 7, \( K \) is in a cycle \( C \) in \( L \). By Lemma 11, there is a path in \( C \), and therefore in \( L \) (possibly of length zero), from \( X \) to \( K \) through \( LI(X) \). Combining this with the edge from \( K \) to \( H \), we obtain a path in \( L \) from \( X \) to \( H \). This path, by Definition 13, gives rise to an edge in \( L' \) from \( X \) to \( H \), which is then a loopback in \( L' \). By removing all loopbacks from \( L \), we obtain \( B \), whose label loop is \( B' \). Any edge in \( B' \) arises, by Definition 15, in the same way that it did in \( L' \), unless it was an edge leading to \( H \). Thus all edges in \( L' \) remain in \( B' \) except those leading to \( H \); and these are exactly the specifications of the body of \( L' \). □

We can now show a correspondence between loop trees in \( L \) and in \( L' \).

**Theorem 3.** Given a labeled program with a flow graph \( G \), a loop tree \( T \), and a label graph \( G' \), the label loops, as in Definition 15, of all loops in \( T \) constitute a loop tree \( T' \) for \( G' \); and every loop tree of \( G' \) arises in this way from a loop tree of \( G \).

**Proof.** We proceed by induction on the height \( h \) of \( T \). The case \( h = 0 \) is clear, because if \( G' \) has a cycle \( C' \), then \( C' \) has at least one expanded cycle \( C \) in \( G \), by Definition 14; hence if \( G \) is acyclic, then \( G' \) is also. The label loop of any outer loop (strong component) \( L \) of \( G \) is an outer loop \( L' \) of \( G' \) by Theorem 2; any head \( H \) of \( L \) can also be a head of \( L' \), by Lemma 17; and the label loop of the body \( B \) of \( L \), with head \( H \), is the body \( B' \) of \( L' \), with head \( H \), by Lemma 18. The height of a loop tree \( T_L \) of \( L \) is less than \( h \), and, by inductive hypothesis, the label loops of all loops in \( T_L \) constitute a loop tree \( T_{L'} \) for \( L' \). Since this holds for any outer loop \( L \) of \( G \), the collection of all corresponding outer loops \( L' \), each one with its subtree \( T_{L'} \), defines a loop tree \( T' \) for \( G' \).

To show the converse, we proceed by induction on the height of \( T' \), which we again refer to as \( h \). The case \( h = 0 \) is again clear, because if \( G \) has a cycle \( C \), then \( C \) must be an expanded cycle of some cycle \( C' \) of \( G' \), by Lemma 12; hence if \( G' \) is acyclic, then \( G \) is also. If \( L' \) is an outer loop (strong component) of \( G' \), then let \( M \) be the non-trivial strong component of \( G \) which corresponds to \( L' \) by Theorem 2. By this theorem, \( L' \) is the set of all labeled statements of \( M \), or (by Definition 15) the label loop of \( M \). Also by the theorem, there may be trivial strong components of \( G \) which correspond to \( L' \); however, since these are trivial, they do not appear in any loop tree of \( G \). If \( H \) is a head of \( L' \), \( H \) can also be a head of \( M \), by Lemma 17; and if \( B' \) is the body of \( L' \), with head \( H \), then \( B' \) is the label loop of the body \( B \) of \( M \), with head \( H \), by Lemma 18. The height of a loop tree \( T_{L'} \) of \( L' \) is less than \( h \), and, by inductive hypothesis, \( T_{L'} \) arises from a loop tree \( T_M \) of \( M \), each of the loops in \( T_{L'} \) being the label loop of a loop in \( T_M \). Since this holds for any outer loop \( L' \) of \( G' \), the collection of all corresponding outer loops \( M \) of \( G \), each one with its subtree \( T_M \), defines a loop tree \( T \) for \( G \). □

**Theorem 3, like Theorem 2, gives rise to a definition.**

**Definition 16.** Given a labeled program with a flow graph \( G \) and a label graph \( G' \), as in Definition 13, the label loop tree of any loop tree \( T \) of \( G \) is the loop tree \( T' \) of \( G' \) which exists by Theorem 3, whose loops are the label loops of the loops in \( T \).

All this is illustrated in Fig. 12, showing the reduction in rearranger space requirements by using the label graph instead of the original graph. This figure also shows how the loop trees correspond, under this transformation.
The labeled program $G$ (a) has a flow graph with 11 nodes (b), having a loop $L$ containing nodes 4, 5, 6, 9, and 10. The label intervals of $G$ (c) give rise to the label graph $G'$ of $G$, having the loop $L'$ (d). Here $L'$, being the set of labeled statements in $L$, is the label loop of $L$ by Definition 15. Following Theorem 2, let $Y$ be the union of all $LI(J)$ for $J$ in $L$; then $Y$ contains nodes 4, 5, 6, 7, 8, 9, and 10. Thus $Y$ is the disjoint union of $L$, as above, with the nodes 7 and 8, each of which, by itself, is a trivial strong component of $G$, as specified by Theorem 2. The loop trees of $G$ and of $G'$ correspond (e), as specified by Theorem 3.

Theorem 3 and Definition 16 indicate how to build a rearranger capable of processing a large-scale program, whose flow graph cannot fit into memory all at once. Only the label graph, not the original flow graph, is processed. The source program file is now on disk, indexed by file pointers. The starting and ending file pointer of a label interval $LI(X)$ is associated in memory with $X$ as a node of the label graph. These file pointers may be determined by a single pass through the (possibly large-scale) source program.

A disadvantage of this approach arises in constructing the rearranged program from the rearranged graph, since label intervals must now be taken from the disk. Since these are normally not in the original order, the result may be some disk thrashing, which would not occur if the entire program were in memory at once. Of course, a two-pass assembler could also be speeded up if the entire source program were in memory at once, since the source file would not have to be read twice from disk.
11. Some possible concerns

The author is grateful to the reviewers for pointing out the following possible areas of concern, in addition to that of scalability treated above.

11.1. Program equivalence and code rearrangement

One area of concern is that of rearrangement of code. The discussion of Fig. 11 can be easily misunderstood by those who remember that, for example, every Turing machine can be implemented by a structured program. Since every program can be simulated by a Turing machine, it is a trivial observation that every program is equivalent to a structured program (indeed, one with a single loop). However, as we saw in Section 8 above, rearrangement of code provides more than mere equivalence. The new flow graph is identical to the original flow graph; every execution sequence of one is identical to an execution sequence of the other. There is, therefore, no speed penalty associated with rearrangement of code, as there is with Turing machine simulation.

11.2. Object-oriented design

Our use of procedural examples may lead to a concern about whether this work is relevant to the modern world of object-oriented design. Of course, most object-oriented designers are already writing structured programs; indeed, Java has no goto statement, so spaghetti code is impossible to write in Java. Even in C++, which has the goto, most people avoid it, when writing object-oriented code. One answer to this is that loop trees have to do entirely with control, whereas object orientation has to do entirely with data. No matter how many object references there are, or are not, in a program, its loop trees remain unchanged if its control remains unchanged.

There are, however, implications of loop trees for writing programs in an object-oriented language. Consider again Fig. 6(a) above; this looks like two nested loops, although we saw, in Section 4.2 above, that it is actually two sequential loops, with a goto statement. In Java, there is no way to express this, since Java does not have the goto; hence rearrangement of Java programs does not always produce syntactically correct Java programs, as shown in Fig. 13(a). Python also does not have the goto, and yet this rearranged program may be expressed in Python by using the else statement after a for loop, as in Fig. 13(b). Here, if the first for loop exits normally, the else is executed, and 0 (meaning false) is returned. If this for loop exits through break, however, control goes past the else, to the second for loop.

12. Non-rooted graphs

We saw in Section 3 above that meaningful flow graphs are always rooted. Coincidentally, our proof of Theorem 1 applies directly only to rooted graphs. Nevertheless, it may be generalized to all graphs, as we now show.
In a non-rooted graph, Lemma 8 does not necessarily hold. Consider, for example, two disjoint cycles, one of which contains the start node; then the other one does not contain a head, in the sense of Definition 8. Nevertheless, a generalization of Theorem 1 still holds, because nowhere in the proof do we use the fact that every loop has a head in this sense.

Specifically, we define a generalized semantic loop tree, in the same way as in Definition 9 except that the head of a loop \( L \) can now be an arbitrary node of \( L \). For a rooted graph, this is unnecessary, since there is always at least one ordinary semantic loop tree by Lemma 8. For a non-rooted graph, however, we use the proof of Theorem 1, exactly as it stands, to show that it holds for generalized loop trees as well. (The generalization is actually needed only at the top level, since the body of any loop is a rooted graph by Lemma 10.)

13. Alternate notations and related work

Terminology for self-loops, for rooted graphs, and for component graphs is not standardized. Our definition of a self-loop follows [32, p. 319], although a self-loop in sometimes called a sling, or even, simply, a loop [23]. To distinguish such loops from our own use of the term “loop” (which follows [18, pp. 57–58]), we referred, in [29], to a self-loop as a one-node cycle. Our definition of a rooted graph is based on that of [19, p. 188]; also see [25, p. 539]. Sometimes a rooted graph is called connected [10, p. 73], and the root, or start node, is sometimes called a source [17, p. 98]. Our definition of a component graph is based on that of [11, p. 554]; it is also known as the reduced graph [16, p. 20], the superstructure of \( G \) [13, p. 64], or the (acyclic) condensation of \( G \) [18, p. 34].

Flow graphs have been studied extensively. Sometimes they are defined specifically for assembly language programs [5,7,41]; and sometimes their nodes are restricted to outdegree no more than 2 [7]. Each node of a flow graph is sometimes taken to be a basic block, rather than a single statement, as in [3, p. 528]; here a basic block is a single-exit, as well as single-entry, sequence of statements (our Definition 13 is an extension of this idea). Call graphs have also been studied extensively [4,33]. We have not considered possible applications of our work to dependency graphs for sets of equations; in such a graph, “each node represents a variable; each directed edge (\( m, n \)) represents the dependence of \( X_m \) on \( X_n \) (i.e., the occurrence of \( X_m \) on the right-hand side of the equation for \( X_n \)” [34, p. 286].

The basic idea of a syntactic loop tree is described informally, for while loops, in [35, p. 25]. The idea of a “trivial region,” meaning what we are calling a trivial strong component, is defined by Lee, Marlowe, and Ryder [24, p. 943]. These authors define a strong component as what we are calling a non-trivial strong component, and therefore speak of “nodes not participating in a strong component” (see also [34, p. 301]). The observation that a (semantic outer) loop is a strong component of a flow graph is made in [35, p. 22].

The idea that “when two . . . loops have the same header, but neither is nested in the other, they are combined and treated as a single loop” is noted in [3, p. 605]. Loopbacks are sometimes also called latches [18, p. 58]. Construction of a topological sort of a component graph, as at the start of the proof of Theorem 1, is made in [27, p. 188].

Many authors have decomposed strong components into smaller parts, although not in the precise way set forth here. The earliest such decomposition is in [26], revived and expanded as “wheels within wheels” ([22]; see also [9]). Tarjan’s dominator trees ([38]; see also [1]) are another similar construction, as are his decompositions of weighted strongly connected graphs [39]. Other decompositions of such a graph involve breaking it up into biconnected, triconnected, etc., components [12,20] or unilaterally connected or weak components [40, pp. 570–574]. We have further described the relationships among these constructions in [29].

One related construction deserves special mention since, although it does not involve strong components, it directly addresses the same problems that we do. This was introduced in [5] and further extended in [19], and is today referred to as the method of derived sequences. It is a bottom-up decomposition, with innermost loops being identified first, in contrast to the top-down decomposition described in Definition 9. It works only for reducible graphs, as these are defined in [5]. An irreducible graph may always be transformed into a reducible graph, using the techniques of [41] and [31]; however, the applicable transformation might involve replicated code, and might also involve new variables and statements.

We have given our own definition of reducibility, in terms of loop trees (Definition 10); in [29], we show its equivalence to that of [5] and [19]. There is a similar result in [18, p. 101]: “A flow graph \( G \) is reducible if and only if all strongly connected regions of \( G \) are single-entry.” A strongly connected region is here the induced subgraph of any strongly connected set of nodes whatever; the number of cycles (and thus the number of strongly connected regions) in a graph can be exponential in the number of vertices [32, p. 348].
References


