AN ABSTRACT OF THE THESIS OF

RICHARD EARL MCALISTER for the degree of MASTER OF SCIENCE

in Computer Science presented on January 31, 1980

Title: Knots: A Measure of Program Complexity

Abstract approved:

Dr. Paul Cull

Most measures of program complexity gauge either textual or control flow attributes of a program. A recent addition to the field of complexity measures, the knot metric, is a function of both these attributes. A knot measurement reflects the degree of control-flow tangle in a program's listing. This thesis discusses and proves four functional properties of the knot measure.

1. Calculation of a program's knot content is fast with respect to the number of branches in a program. A worst-case optimal algorithm for computing knots is quadratic in time and linear in space.

2. The complexity of a program can be reduced by rearranging groups of statements in a manner that retains the program's function yet lowers its knot content. The problem of finding an arrangement with the fewest knots for any arrangement of a program is probably difficult or NP-complete, but approximation methods are fast and often find the minimum knot arrangement.

3. A direct relationship exists between the types of knots in a program text and the structuredness of that program. This leads to an easily testable, sufficient condition for unstucturedness. Thus unstuctured programs may be detected without graphically reducing the control structure to structured programming conventions.

4. An empirical investigation of a set of FORTRAN programs, testing for their knot content, rearrangement characteristics, cyclomatic number, and program length, demonstrates the practicality of the knot measure. Most programs benefited from rearrangement, and a fast, heuristic algorithm was effective in finding a program text ordering with minimal knot content. Furthermore, the knot content of a program is dissassociated from two other measures of complexity, cyclomatic number and program length. Knots must measure some aspect of complexity missed by those measures.

Overall, the knot metric is an effective, and efficient means for detecting, reducing, and controlling some attributes of software complexity.

Knots: A Measure of Program Complexity

by

Richard Earl McAlister

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed January, 1980

Commencement June, 1980

APPROVED:

Redacted for Privacy

Professor of Computer Science
in charge of major

Redacted for Privacy

Chairman of Computer Science

Redacted for Privacy

Dean of the Graduate School

Date thesis is presented January 31, 1980

Typed by Richard McAlister

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

KNOTS: A MEASURE OF PROGRAM COMPLEXITY

INTRODUCTION

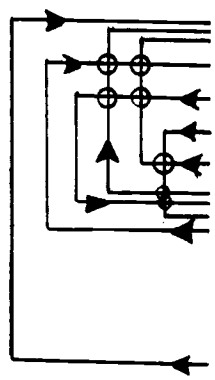
A number of measures of program complexity have been proposed. In this thesis, I will investigate the knot metric recently proposed by Woodward, Hennel, and Hedley. (30) I will discuss algorithms for computing the number of knot in a program and give an algorithm which is worst-case optimal in both space and time. I will specify the rearrangements of program statements which preserve functionality and yield the smallest number of knots. I will argue that the problem of finding this rearrangement is probably NP-complete by showing that a slight generalization of the problem is NP-complete. I will discuss and prove theorems about the relationship between knots and structured programming. Finally, I will report on some empirical studies which indicate the number of knots is independent of other measures of program complexity, and that finding the rearrangement which gives the fewest knots was easy for the programs studied. In general, I will build a functional and empirical justification for the use of knots as a program complexity measure.

PROGRAM KNOTS

The knot metric originates in a technique used by programmers to display control flow on program listings. Some programmers draw lines in the left-hand margin of a program listing between statement branches and entries. (Figure 1) The resultant path diagram graphs the flow of the control flow in a program, outside of implied statement sequencing. Woodward et al. (30) observed that the code is less difficult to follow when the control paths do not cross on the listings. They suggested that the number of these crossings or "knots" is a good measure of program complexity.

A more precise definition of the crossing of two control flow paths requires an association of statement position in the program listing and statement branching. The definition of a knot graph to represent a program best combines these attributes of statement location and succession for analysis. The knot graph is a directed graph in which each node corresponds to a statement in the program. Every node is labeled with the line number of that statement. The directed edges of the knot graph represent transfer of control from statement to statement during program operation.

> A knot graph, $G(N,E,L)$, of a program $P=\{p_1,p_2,p_3,\ldots,p_m\}$ where p_i is a statement, consists of a set of nodes $N=\{n_1,n_2,\ldots,n_m\}$, a set of edges $E=\{e_1,e_2,\ldots,e_s\}$ such that $e=(n_i,n_j)$ is in E when $p_i \rightarrow p_j$ is a transfer of control in P, and a one-to-one function $L:L(N)\rightarrow\{1,2,\ldots,m\}$ such that if p_i is on line number j, $L(n_i)=j$.



```
Line number   Text
     1          LOGICAL FEM(8), MALE(8)
     2          READ(5,6) IGIRL,(FEM(I),I=1,8)
     3        9 READ(5,6) IBOY,(MALE(I),I=1,8)
     4          DO 8 I=1,8
     5          IF(FEM(I)) GO TO 7
     6          IF(.NOT. MALE(I)) GO TO 8
     7          GO TO 9
     8        7 IF(.NOT. MALE(I)) GO TO 9
     9        8 CONTINUE
    10          WRITE(2,10) IBOY
    11        6 FORMAT(10X,I5,8L1)
    12       10 FORMAT(10X,I5)
    13          GO TO 9
    14          STOP
    15          END
```

Figure 1: Control flow paths producing knots.
Program is from (19).

In the program text, a path drawn in the left-hand margin connects the line of a statement which jumps to the line of the statement to which the jump goes. Two such paths cross or knot if each path has exactly one end-point between the end-points of the other path.

In the knot graph, two labeled nodes represent the line-numbered end-points of a jump between statements in the program text. If, for any two jumps, exactly one of the node labels of each of the representative knot-graph edges is exclusively between the two node labels of the other path, a knot occurs.

Given a knot graph, G(N,E,L), a pair of edges from E, $e=(n_a,n_c)$ and $e'=(n_b,n_d)$, with labeling function, L, such that $L(n_a)=a$, $L(n_b)=b$, $L(n_c)=c$, and $L(n_d)=d$, are knotted if and only if

$a<b<c<d$ or $d<c<b<a$
or $c<b<a<d$ or $d<a<b<c$
or $a<d<c<b$ or $b<c<d<a$
or $c<d<a<b$ or $b<a<d<c$.

The knot number or knot count of a program is the number of knots in the program's knot graph. That is, if every edge in a program is compared to every edge once, the number of these pairs that knot is the knot number of a program.

For two edges, $e=(n_a,n_c)$ and $e'=(n_b,n_d)$ from a knot graph G(N,E,L), the knot function over e and e' is X(e,e'):

$$X(e,e') = 1 \text{ if } e \text{ and } e' \text{ knot,}$$
$$= 0 \text{ otherwise.}$$

The knot number of a program P with knot graph G(N,E,L) where $E=\{e_1,e_2,e_3,\ldots,e_s\}$ is a summation of the knot function over the set of edge pairs from E. The knot number is designated KNOT(P) such that

$$KNOT(P)=\sum_{i=1}^{s-1}\sum_{j=i+1}^{s} X(e_i,e_j).$$

The knot may be defined in terms of two other graph representations: the labeled flow graph, and the overlap graph.

The flow graph is similar to the knot graph in that nodes represent statements and edges represent control flow of a program. Each node in a flow graph can, however, represent more than one statement from a program. The sets of statements which make up a flow graph node are program blocks in the program text. A program block is a sequence of textually adjacent statements of which the first statement is the only statement where some entry to that statement is by jump, and the last statement is the only statement which is a jump. Control transfer from statement to statement inside a program block is only by natural succession. Figure 2 shows a program split into program blocks and the representative flow graph.

The order or labeling of program blocks as nodes in a labeled flow graph is a function of their position in the program text. With this labeling, a labeled flow graph is a directed graph representation of a program text from which knots in the program may be computed using the knot function previously defined over the labeled edges.

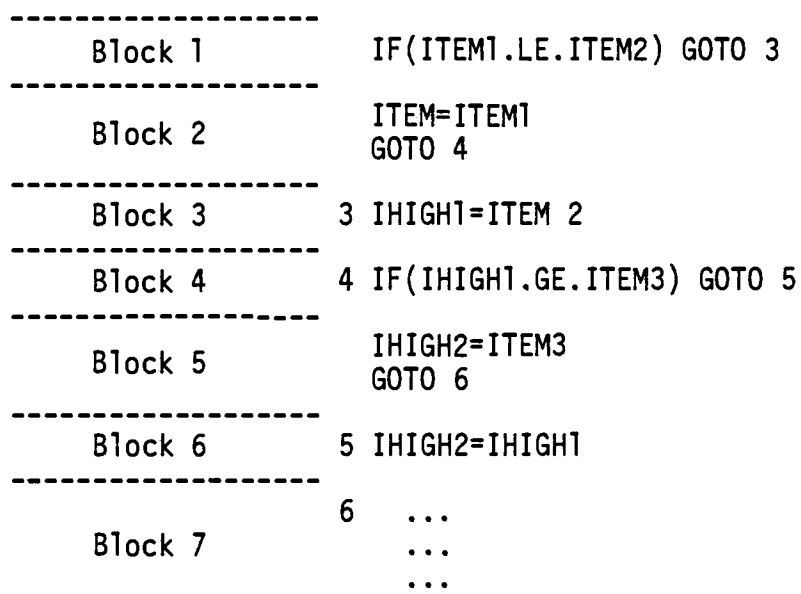


Figure 2: A program fragment partitioned into program blocks. Program taken from (19).

A labeled flow graph, $G_f(N,E,L)$, of a program $P=\{b_1,b_2,\ldots,b_m\}$ where each b_i is a program block in P, consists of a set of nodes $N=\{n_1,n_2,\ldots,n_m\}$, a set of edges $E=\{e_1,e_2,\ldots,e_s\}$ where, if $b_i \rightarrow b_j$ is a program block sequence of control, then $e=(n_i,n_j)$ is in E, and a one-to-one function $L:L(N)\rightarrow\{1,2,\ldots,m\}$ where if b_i is ordered in the program text as the jth program block, then $L(n_i)=j$.

Woodward et al., in the original knot paper (30), used the labeled flow graph as the knot function basis. In Section 2, I will discuss use of that graph for computing knots. One reason for the selection of the flow graph as representation for computing knots was that McCabe based the cyclomatic number on the flow graph.(23) The cyclomatic number is another complexity measure which will be mentioned often. As an aid to later references, I will define the cyclomatic number here.

The cyclomatic number of a program is the number of linearly independent circuits in its flow graph model. This number is one more than the number of branchings in a program.

The third useful type of graph is the overlap graph. (3) This type differs from the previous two in that its nodes represent control flow edges. An edge in a knot graph can be represented as an ordered pair of labels, those of the nodes incident upon it: $(L(n_i),L(n_j))$. In the overlap graph, this ordered pair is the label for a node. In this sense, the overlap graph is a labeled edge-graph of a knot graph. The edges of the overlap graph do not correspond to an edge-graph, however. An edge between two nodes of an overlap

graph exists only if the edges represented by the two nodes knot.

> An overlap graph, $G_o(\underline{N},\underline{E},\underline{L})$, defined in terms of a knot graph, G(N,E,L), and a knot function $X(e_i,e_j)$ consists of a set of nodes $N=\{\underline{n}_1,\underline{n}_2,\ldots,\underline{n}_m\}$ such that for each e_i in R, there is one $\underline{n}_i$ in $\underline{N}$ where the labeling $\underline{L}(\underline{n}_i)=(L(n_j),L(n_k))$ for each $e_i=(n_j,n_k)$, and the edge $\underline{e}=(\underline{n}_i,\underline{n}_j)$ is in $\underline{E}$ if and only if $X(\underline{n}_i,\underline{n}_j)=1$, that is the labels of the two nodes knot.

Cook's (13) discovery of the relationship of overlap graphs to the knot concepts will be discussed in Section 2 in conjunction with the knot counting algorithms based on this graph.

Within these many formal representations of a knot, the intuitive feel for the association of a knot in a program and the complexity of a program gets lost. I do not wish to omit the intuitive aspects of the knot measure. After all, the knot itself developed from practice, not theory.

A working rule used by the drafters of electrical schematics is a basic example of this association. A schematic is more difficult to read when lines interconnecting components cross. When lines do not cross, components can be visually grouped and the design studied in a modular fashion. Intersection connections also inhibit rapid tracing of individual connections. Minimization of crossovers leads to the most useful and readable schematics.

The knot metric is an application of this concept to program text. A knot indicates a crossover in control flow in the program text. The greater the number of crossovers, the less the program can be visually modularized. When tracing control paths through the

program, a knot indicates that some other path branches around the one being currently traced. The crossing path must be remembered until it can be checked. The crossovers defined as knots in programs reflect the same type of complexity as crossovers in schematics.

The nature of the knot measure is further revealed by the software attributes from which it derives. The primary attribute upon which knots depend is control flow. Knots are created by branching within a program, but knots relate more to the quality of control flow than to the quantity. A program with three properly nested DO-loops has three branches and no knots. An added branch from outside the nest into the deepest loop only increases the branch count by one, but causes three knots. The number of knots is unaffected by well-ordered branches in any quantity, but grows quite rapidly when a tangle develops. Knots depend on the interweave rather than the amount of control flow.

The knot concept allows a quantification of control flow tangle because of its "text space" (8) dependency, the other software basis. Knots depend on the relative position in the program text of statements which branch. In Section 3, for example, I describe methods for reordering statements thereby altering a program's knot count without altering the program's function. Ultimately, the knot count depends on how that branching is distributed in the program's text, irrespective of the control flow characteristics of a program.

The knot metric, then, combines two software bases: control flow and program text. Control flow, evaluated from its textual representation produces program knots. The measure combines the intentions

of previous complexity measures. McCabe (23), Gilb (13), Myers (25), and Dijsktra (8) each conjectured that the prime constituent of complexity is program branching. Each defined or suggested a different measure of that attribute. Halstead (16) and Gordon (14,15) assumed complexity to be a textual factor. Measures of operator, operand, and program length were specified. Hansen (17) applied some textual basis to control flow by combining the branch count with the number of operators in a program. Whether by argument or data, each of the measures demonstrated some ability to reflect the complexity of a program. (4,9,10,11,22) If the results of each of these types is reliable, clearly complexity is based in both control flow and textual attributes of software. (17) Not only do knots use this base, but the algorithmic independence of the measure in that use implies the property of software complexity which differentiates it from computational complexity. (14) Thus knot number appears very justifiable as a measure of program complexity.

Although well supported, the measure requires the empirical evidence necessary for proof of an association with complexity. (14,15,22,34) As a means to that end, this paper covers the groundwork for implementation of the knot metric in software development systems. The second section examines the algorithmic complexities of determining the number of knots in a program. The third section analyzes the metric in its use as a tool for reducing the complexity of a program. The result that the knot can be used to create clearer programs, without semantic or syntactic program modification, emphasizes the relationship of the measure with complexity. The

fourth section covers the relationship of knots and structuredness in programs. The results detail the association between the modularity of a program and the complexity measure. The fifth section examines data gathered on existent programs for their knot content. The knottedness of these programs is compared with other program attributes, including another complexity measure, the cyclomatic number. The results establish the independence of the knot metric from other complexity measures based on either textual or control flow properties. All four sections build the foundation for the implementation of the knot metric as a software quality tool.

ALGORITHMS FOR COMPUTING KNOT NUMBER

The best algorithm for computing knots is the one which uses the least space, i.e. fewest memory locations, and the least time, i.e. executes the fewest instructions. Other desirable properties of an algorithm are generality and portability. That is, an algorithm should work for all input programs and no special feature of a programming language should be used.

In this section, I will present an algorithm which meets these criteria based on the knot graph representation of a program. As a means of comparison, I will then analyze the original algorithm by Woodward et al. (30), based on a flow graph representation of a program. Lastly, I will demonstrate that Cook's algorithm (3) based on an overlap graph representation of a program is equivalent to that based on the knot graph, with minor modification.

As a basis for proof of an worst-case optimal algorithm in computational complexity, an input attribute must be chosen to gauge the algorithm. Computational complexity is normally assessed as a function of the "size" of the input to the algorithm. In evaluation of algorithms which use a graph representation of program flow as input, two candidates arise: Number of program statements or graph nodes, and number of program branches or graph edges.

Only one of these choices is adequate for measuring knot numbering algorithms, the number of branches. The choice was not made simply because the knot is defined in terms of the control flow edges. If the number of statements in a program text would provide a measure

of program size, the number of statements must force an upper bound on the number of edges in a program. However, the amount of branching in a program is independent of the number of statements within it. A computed GOTO may have any number of repetitions of destination labels in the FORTRAN programming language. A program's knot graph may, therefore, have parallel edges. In any graph allowed to have parallel edges, the number of edges is independent of the number of nodes. The knot function is edge dependent, and, as the number of edges in the knot graph are not bound by the number of nodes, the "size" of a program input to a knot counting algorithm must be measured by the amount of branching of the program, that is, the number of edges in its representative knot graph.

> The size of a program input to a knot counting algorithm is the number of possible statement-to-statement successions of control, either implied or explicit, in a program as inferred from its text. From the program's knot graph representation, G(N,E,L), the size of a program is measured by the variable $\varepsilon = |E|$.

The knot-graph edge, or program branching, is also the basis for the space usage measure and time usage measure for the knot-graph algorithms. The unit of space use or cell is the storage of a single knot-graph edge. The space complexity of the knot algorithm is, then, the number of knot graph edges stored during computation as a function of the number of knot-graph edges in the program text input for that computation.

The basic operation upon which time complexity is judged is the comparison of two knot graph edges and their associated node

labelings. An incremental operation of a knot counting routine is access and comparison of the end-point line numbers of two control flow edges from a program. Time complexity is the number of comparisons on knot-graph edges as a function to the size of the program input.

The following is an algorithm for computing the knot number of a program:

Problem:

Compute the knot number of a program from its knot graph representation.

Class of Algorithms:

Algorithms which compute knot numbers from knot graphs.

Basic Operation:

Comparison on two knot-graph edges.

Space Cell:

A knot-graph edge, $e=(n_i,n_j)$.

Algorithm:

Input - Program control flow edges and labelings represented by a knot graph, G(N,E,L).

Output- Knot number of the program input, K.

A: Store for B only edges $e=(n_i,n_j)$ from E such that

$$1 < |L(n_i)-L(n_j)| < |N|-1.$$

B: Input from A the values $e_1,e_2,e_3,e_4,\ldots,e_s$ as labeled edges.

```
1.  k=0; i=0;
2.  while i ≤ s-1  do
3.      j=i+1;
4.      while j ≤ s  do
5.          k=k+X(e_i,e_j);  /* X(e_i,e_j) is the knot function */
6.          j=j+1;
7.      end;
8.      i=i+1;
9.  end;
```

Space usage

The storage of the algorithm is ε', the set of edges whose incident node labels differ by at least two and at most $|N|-2$. This set of edges will be referred to as E'. The size of E' is ε'. For ε' to be the minimal storage requirement of all knot counting algorithms, every edge in E' must be capable of causing a knot and every edge from the knot graph which is not in E', (E-E'), must be incapable of causing a knot.

The latter claim is immediately obvious from the definition of a knot. An edge $e=(n_i,n_j)$ can knot with another edge $e'=(n_i',n_j')$ only if $L(n_i')$ or $L(n_j')$ is an integer value exclusively between $L(n_i)$ and $L(n_j)$. If there is no integer value between $L(n_i)$ and $L(n_j)$, then the edge e cannot be knotted by any other edge. The other edges not contained in E' are those where $|L(n_i)-L(n_j)| = |N|-1$. The maximum labeling of the nodes of the knot graph is $|N|$, so these edges are those between the first statement line and the last statement line. No edge can branch to or from any node outside this range, so no edge can knot this type of edge. Therefore, no edges in the set E-E' can cause a knot.

In consequence of the previous discussion, the first claim is also true. A knot graph, G(N,E,L), for any edge $e=(n_i,n_j)$ such that $1 < |L(n_i)-L(n_j)| < |N|-1$, must have at least one node n_i' such that:

$$L(n_i) < L(n_i') < L(n_j) \quad \text{or} \quad L(n_j) < L(n_i') < L(n_i),$$

and at least one node n_j' such that:

$$L(n_j') < L(n_i) \quad \text{and} \quad L(n_j') < L(n_j),$$

$$\text{or} \quad L(n_i) < L(n_j') \quad \text{and} \quad L(n_j) < L(n_j').$$

For any edge e in the set E', an edge between n_i' and n_j' would knot it. Obviously, that edge would also be in E'. The argument implies that every edge in E' can cause a knot and can be knotted. Therefore, at least all the edges in E' must be evaluated for the knot count. The difference in size between E and E' is significant because a program usually will have many more non-branching statements than branching ones. The edges between textually adjacent statements will greatly outnumber those between textually non-adjacent statements.

The result of these two claims is that the knot-graph algorithm is space optimal. The order of space complexity of this algorithm is $O(\varepsilon')$.

Time complexity

The upper bound on the number of knot functions or basic operations of the knot algorithm is $|E'|(|E'|-1)/2$ or $\varepsilon'(\varepsilon'-1)/2$. The value is the number of operations the algorithm will make in terms of the edge set of a knot graph, which is the size reference of the input program. This function of ε' is a result of the iterations of the nested DO-loops.

The lower bound of the algorithm is the greatest number of operations the algorithm must perform for any input set size. The program in Figure 3 has a knot-graph representation with $\varepsilon'(\varepsilon'-1)/2$ knots. The generic type of graph with this knot number is a knot graph in which, for every n_i in N, there is an edge $e=(n_i,n_{i+m})$ in E such that $L(n_i)=i$ for all $1\leq i\leq|N|$, for $m\geq 2$, and $|N|=2m$. All edges e are in E' and each knots every other edge in E'. The total knot

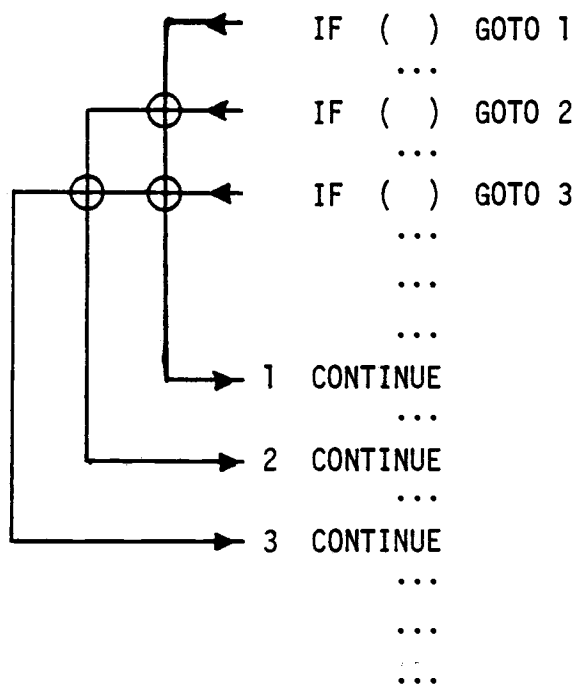


Figure 3: A maximally knotted program form.

count is thus $\varepsilon'(\varepsilon'-1)/2$.

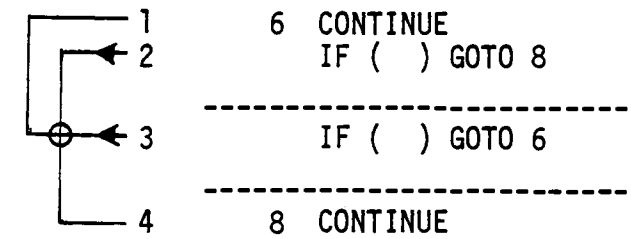
The existence of a knot graph with this knot number indicates that a knot counting algorithm must compute at least $\varepsilon'(\varepsilon'-1)/2$ knot functions or edge comparisons. The upper bound on the knot counting algorithm states that the algorithm always computes at most that many edge comparisons. The algorithm is therefore optimal at $\varepsilon'(\varepsilon'-1)/2$ operations. The above results suffice as proof for the knot counting theorem.

Theorem: The given knot number algorithm correctly computes the knot number of any program using $O(\varepsilon')$ space and $O(\varepsilon'^2)$ time. This algorithm is worst-case optimal in both space and time.

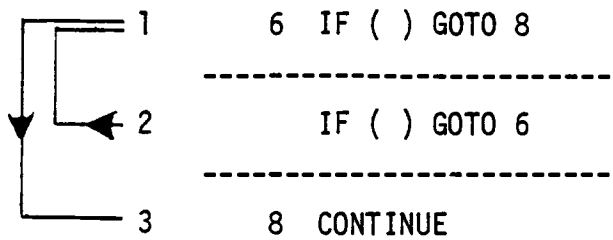
Other algorithms

Woodward et al. (30) produced the first knot enumeration algorithm. The computation was based on the flow graph representation of a program with labeling to define the text locations. Nodes of the labeled flow graph indicate sets of statements as program blocks with labels locating blocks in the program text. The edges of the flow graph represent control flow transfer between program blocks.

The choice of a labeled flow graph representation of a program for computation of the knot number results in the need for an upper and lower bound on the knot number for a program. A program block may contain one or several statements. Figure 4 shows the difference in the possibilities for knots between the different size program blocks. In Figure 4 (I), the control flow is knotted; In Figure 4 (II), the control flow is not knotted. As the node labelings in the labeled flow graph cannot distinguish between these two program block types,



(A)



(B)

Figure 4: Examples of the possible different knot contents between single (B) and multiple (A) statement program blocks.

an upper and lower bound on the knot number of a program is defined. The result is an upper bound knot function and a lower bound knot function. The upper bound function assumes all program blocks are multiple statement. The lower bound function assumes all program blocks are single statement.

> Given a labeled flow graph, $G_f(N,E,L)$, and two edges $e=(n_i,n_j)$ and $e'=(n_i',n_j')$ from E, the edges produce a knot in the lower bound if, for $L(n_i)=a$, $L(n_j)=c$, $L(n_i')=b$, and $L(n_j')=d$,

	a<b<c<d	or	d<c<a<b
or	c<b<a<d	or	d<a<b<c
or	a<d<c<b	or	b<c<d<a
or	c<d<a<b	or	b<a<d<c.

The lower bound knot is equivalent to the knot-graph knot definition. The lower bound knot number is the number of unique edge pairs from E which are knotted under this definition.

> Under the same labeled flow graph specifications as above, the edges produce a knot in the upper bound if

	a<b<c<d	or	d<c≤a<b
or	c≤b<a<d	or	d≤a<b<c
or	a<d<c≤b	or	b<c<d≤a
or	c<d≤a<b	or	b<a<d<c.

The upper bound knot is defined with the same conditions as the lower bound knot with some added equalities. The multiple statement program blocks must have edges in the form shown in Figure 5 to create the upper bound knots. The upper bound knot number is the number of unique

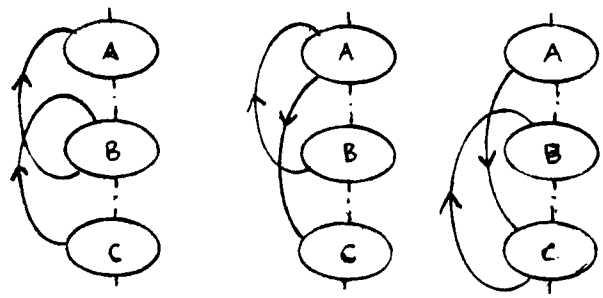


Figure 5: Three types of edge configurations for upper bound knots in the program blocks of a flow graph.

edge pairs from E which knot under the definition of an upper bound knot.

The procedure outlined in (30) computes the knot numbers of a program with the adjacency matrix representation of the labeled flow graph. Both the rows and columns of the matrix are ordered in the program block ordering of the program. The indices of the matrix are thus the labeling function. In the adjacency matrix, and edge in the flow graph of a program with adjacent nodes p and q as labels, is represented by a 1 entered in location (p,q). For an edge (p,q), the lower bound on the number of knotted edges to (p,q) is a summation of the shaded entries in Figure 6. The upper bound is the lower bound added to the number of entries in the shaded areas of Figure 7. Figure 6 shades those nodes exclusively between p and q with possible adjacency to nodes exclusively outside of the p-q interval. Figure 7 adds those nodes with possible adjacencies corresponding to the "equals" conditions in the upper bound definition. When these areas are checked for every edge in a flow graph, the resulting knot number is twice the number of knots in the program represented by the flow graph, in both upper and lower bound numbers.

The Woodward et al. algorithm has two basic faults. The first is the loss of accuracy with the upper/lower bound count. The output pair does not enumerate the knots, but sets an error range in which the actual knot number occurs. In (30), the pair of values are favorably compared with the Myers extension to the cyclomatic number.(25) The comparison is ill-advised. The Myers extension actually adds information about the cyclomatic number. It details the number of extra

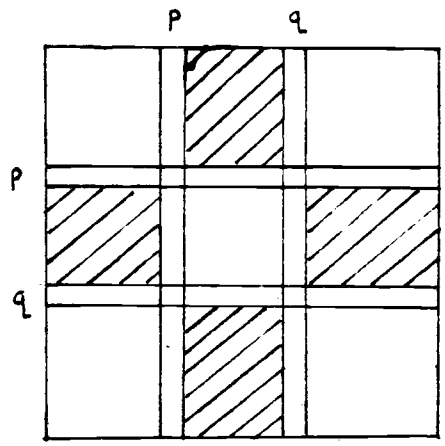


Figure 6: Lower bound knot locations on adjacency matrix for the flow graph representation.

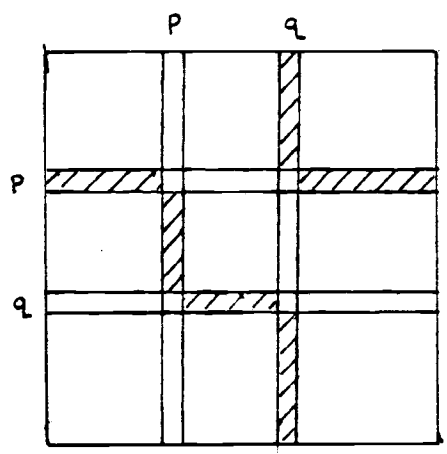


Figure 7: Upper bound knot additions on adjacency matrix for the flow graph representation.

predicates in the conditionals for branching of the program. The upper/lower knot number bounds, on the other hand, introduce an uncertainty into the number of knots in the program. The difference in the information content is also apparent in the relationship between the members of each pair. The integral difference between the members of an ordered Myers' extension is the number of extra conditions found in the program that cause branching. The difference between the upper and lower knot number bounds carries only the level of inaccuracy of both. The only remaining similarity between the two types of measure is that they are both based on the flow graph; the relationship does not imply any innate worth in the upper/lower knot number bounds.

The inaccuracy of the upper/lower bounds on the knot number causes another problem. By specifying a degree of vagueness to the algorithm's output, refinement of knot type is effectively disabled. If the knot number produced by the procedure is in doubt, extension to a specific knot characteristic is not warranted. Refinements are necessary for structuredness determination. (Section 5)

The second inadequacy of the flow graph method is inefficiency. The algorithm is a least twice as complex in time usage as necessary; the procedure computes twice the number of knots found in a program. The more efficient algorithm computes only the minimum number of knots for any program. The space usage of the algorithm is also somewhat inefficient. Although the adjacency matrix is rather sparse for the average program and efficient means exist for storage of a sparse matrix (29), more space than necessary is used. The only edges necessary for knot computation are those between non-adjacent statements.

In the flow graph representation, information must also be retained about edges between adjacent program blocks, as they result in upper bound knots. An optimal algorithm does not require storage of these edges.

Although the flow graph representation is unsuitable for calculation of program knots, the third program representation, the overlap graph, works quite well. The algorithm for computing knot number due to Cook (3) based on the overlap graph representation (with a slight modification by myself) is entirely equivalent to the knot graph based algorithm.

As previously described, each node in an overlap graph corresponds to a branch in a program. The node label is an ordered pair of line numbers for the exit and entrance of that branch. As the edges of an overlap graph indicate that the nodes incident on the edge knot, a knot numbering algorithm based on this graph builds and enumerates the edge set from the node set. That is, a knot counting routine actually creates the overlap graph of a program from the labeled edges of the program.

Figure 8 shows a program and its overlap graph. The number of edges in the graph is the knot number of the program. Cook's knot number algorithm produces these edges from the labeled statements of the program text, the line numbers. In general, the procedure compares the label set of each node of the overlap graph to every other node in the node set. If the pair of labels knot as defined by the knot function, an edge is created between those two nodes. The number of edges created is the knot number of the program represented

```
 1     10 READ(5) EMPID,HOURS,SRATE,ORATE
 2        IF(EMPID.EQ.77777.0) GOTO 60
 3        IF(HOURS.GT.40.0) GOTO 50
 4        WAGE=SRATE * HOURS
 5        IEMPID=EMPID
 6     30 WRITE(6) IEMPID,WAGE
 7        GOTO 10
 8     50 A=ORATE * (HOURS-40.0)
 9        B=SRATE * 40.0
10        WAGE=A + B
11        GOTO 30
12     60 STOP
13        END
```

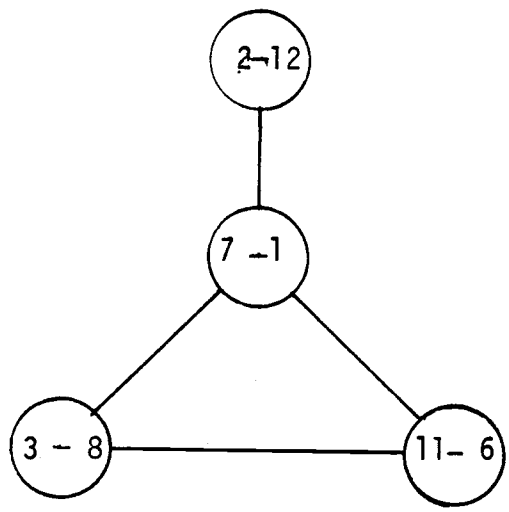


Figure 8: Program and representative overlap graph.

by the overlap graph.

Clearly, in time complexity, the overlap algorithm is equivalent to the knot-graph algorithm. The node set of the overlap graph is the same as the edge set of the knot graph in both size and labeling. The label comparison of the overlap graph nodes is identical to the label comparison of the knot graph edges. The two algorithms must perform the same number of basic operations for every input program, and are, therefore, equivalent algorithms.

The space complexity of the overlap algorithm is slightly greater than the knot graph algorithm. All branches of the program are stored in the label node set. The size of the node set of the overlap graph usually will be larger than the edge set of the knot graph algorithm. However, if the storage is modified to match the edge storage of the knot-graph algorithm, the two space complexities are the same. Then, the stored node set of the overlap graph contains only those statement-to-statement transfers of control which cover at least two lines but not those between the first and last statements. The labels of those nodes stored and compared are the only labels which can knot, so the set is sufficient and equivalent to the edge set for the knot graph algorithm.

In general, the computational complexities of a knot numbering algorithm is limited by the necessity of comparing the end-point labels of each branch to the end-point labels of every other branch. This implies that the input set of a knot numbering algorithm may as well be a set of unique edge pairs as the edge set of a knot graph. This idea gives rise to the generalization of rearrangement in the next section.

STATEMENT REARRANGEMENT FOR MINIMUM KNOT NUMBER

Knots are not only a useful measure of program complexity, but also may be used to reduce complexity in many programs. The technique of complexity improvement is called "rearrangement" (3) or reordering. Rearrangement of a program is the process of interchanging sets of statements to produce a lower knot number. For example, in Figure 9, interchanging the set of statements in A produces a lower knot count in B. The content of the program is unaltered, semantically or syntactically. In (30), Woodward et al. briefly described the prospects of rearranging a program text, but Cook (3) properly defined the problem. Most of the terminology defined here is taken from (3) with only slight modification. I have extended Cook's descriptions to strict definitions for analyses of the concepts of rearrangement.

A program text, P, is an ordered set of statements, $\{p_1,p_2,\ldots,p_m\}$, such that, if $i<j$, p_i precedes p_j in the listing of program P.

The program text in Figure 9 is split into groups of statements called segments.

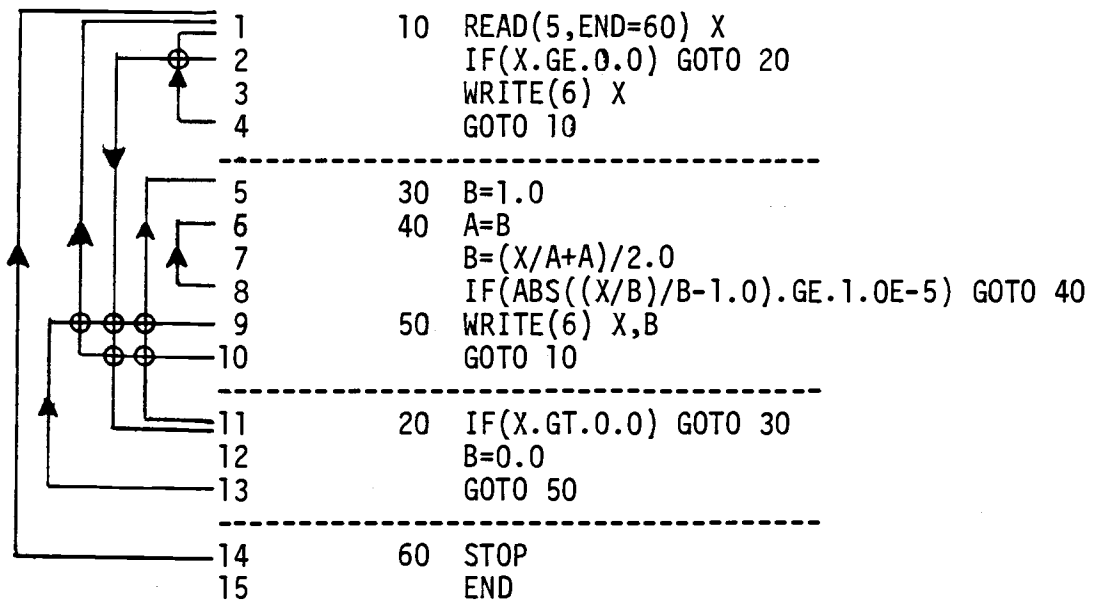
A partition over a program text, $P=\{p_1,p_2,\ldots,p_m\}$, produces an ordered segment set, $S=\{s_1,s_2,\ldots,s_m\}$, such that

$$s_k=\{p_i,p_{i+1},p_{i+2},\ldots,p_{i+t}\}$$

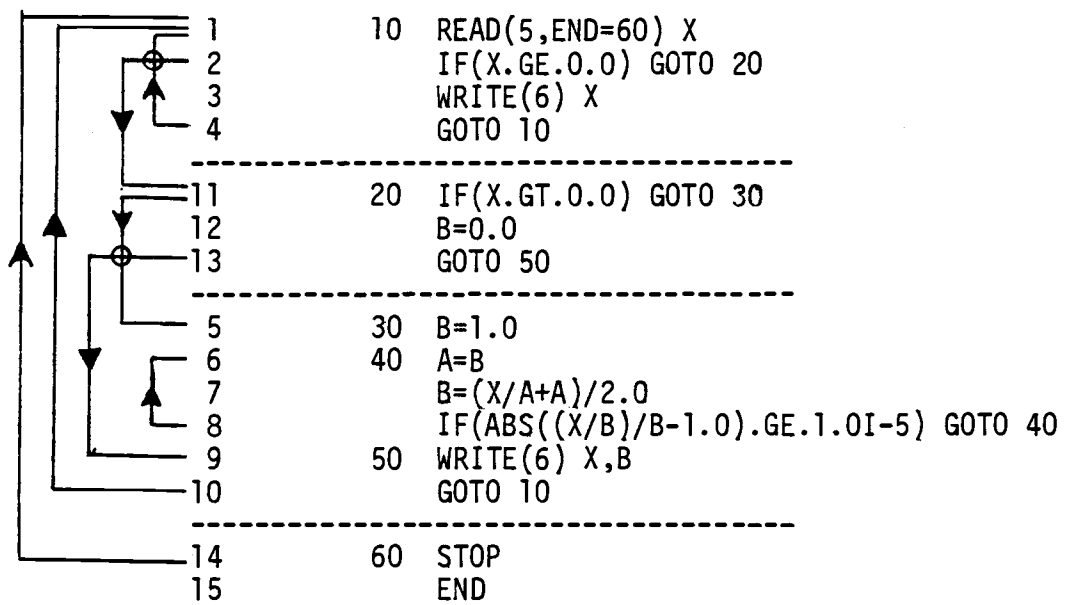
where $t>0$ and for all $2\leq k<m$,

$$s_k=\{p_j,\ldots\} \text{ implies } s_{k-1}=\{\ldots,p_{j-1}\}.$$

The segment definition is the complex way to state that a program can be broken into sequential groups of statements. For rear-



(A)



(B)

Figure 9: A rearrangement of a program for fewer knots.

rangement, the sequences of statements must be self-contained with respect to implied control flow. The flow of control in a program is passed by natural succession (next sequential statement) or explicit branching (labeled jumps). If a program text is rearranged, splitting statements linked by natural succession, the function of the program is disrupted. When rearranging, the natural succession of text must be preserved.

In Figure 9, the statements which allow a valid segmentation of the program are 4, 10, and 13. This type of statement is an explicit branching statement. The special property of the statement is that the statement specifies every succession of control flow explicitly. Such statements are arithmetic IF's and computed and unconditional GOTO's. A segmentation based on explicit branching statements produces independent segments.

> The segments, $s_k=\{p_i,p_{i+1},\ldots,p_{i+t}\}$ of the segment set S, are independent if, for all $1\leq k\leq m$, each p_{i+t} in each s_k is an explicit branching statement, or p_{i+t} is the last statement of the program. If k=1, then i=1 and s_k is the initial segment. If k=m, then p_{i+t} is the last statement of the program and s_k is the final segment.

If the segments are independent, the definition requires that p_i is the first statement of a program or that it immediately follows an explicit branching statement and is labeled for entry. Thus, independent segments result from a split in a program, immediately following an explicit branching statement.

In (3), every independent, initial, or final segment which does

not properly contain an independent, initial, or final segment is an elementary segment. The set of elementary segments is the finest independent-segment partition of program text.

> Independent segment $s_k=\{p_i,p_{i+1},\ldots,p_{i+t}\}$ is an elementary segment if, for all $0\leq s<t$, p_{i+s} is not an explicit branching statement.

Hereafter, in the interests of brevity, a segment is assumed elementary unless otherwise specified.

Rearrangement is a reordering of the segments, other than the initial and final, of a program. For most programs, the designation of elementary segment type is sufficient to allow valid rearrangement, preserving the functional integrity of the program. A reordering of these segments plainly retains the syntactic and semantic structure of a program. But, in some cases, reordering of elementary segments can cause a change in the syntactic structure. An example is shown in Figure 10.

Elementary segments A and C (in Figure 10) cannot change their relative order without disrupting the program. Segment C must follow A. Together, A and C may be rearranged in the program, segment B relocated, or segment D entered between A and C, but the order of A and C must be maintained. The section is not represented as a common FORTRAN construction. In fact, it would be difficult for even a novice to write such tangled code. But a standard FORTRAN compiler would accept it, so the construction is valid.

A simple restriction of DO-loops within relocatable segments is not sufficient either. The program section in Figure 11 has a single

```
-----------------
    DO 20 I=1,10
    IRMX=I/2            Segment A
    GOTO 30
-----------------
10  IRMX=IRMX/4
    GOTO 40             Segment B
-----------------
20  CONTINUE
30  IRMX=IRMX+256       Segment C
    GOTO 10
-----------------
40  CONTINUE
      ...               Segment D
```

Figure 10: Elementary segments inside a DO-loop.

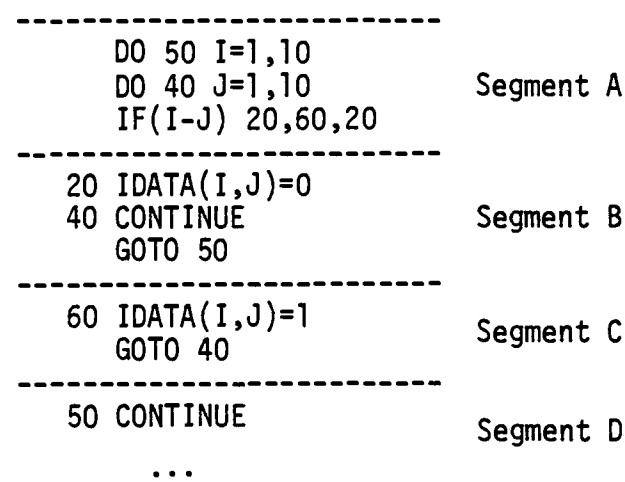


Figure 11: Elementary segments in nested DO-loops.

elementary segment containing the entrance to two DO-loops. Both segment C and segment D must follow segment A, with the added stipulation that C follows B. Segments may be relocated between A and B, B and C, or A and C, but the order of A, B, and C must be preserved. Rearrangement must maintain the order of some subset of elementary segments of a program.

The ordering restriction are, therefore, a set of ordered pairs of segments, (s_i,s_j), such that for any valid ordering of a program, segment i must precede segment j in the text. One subset of these ordered pairs defines the initial segment. That is, if s_I is the initial segment, for every other segment, s_k, there is a restriction (s_I,s_k) so that every segment k follows the initial segment. A similar subset of ordered pairs exists for the final segment.

> The rearrangement restriction set, R, is a set of ordered pairs, (s_i,s_j), of members of the elementary segment set of a program, such that s_i precedes s_j for any valid rearrangement of the program. The initial segment restriction is a subset of R, R_I, such that for some s_i in S, (s_i,s_j) is in R_I for every $j \neq i$. The final segment restriction is a subset of R, R_F, such that for some s_j inS, (s_i,s_j) is in R_F for all $i \neq j$. The set of restrictions not initial or final are the DO-loop restrictions.

In terms of the preceding definitions, rearrangement is a transformation on the order of the segment set of a program, which preserves the rearrangement restrictions. To specify the transformation functionally, a variable z_i is associated with each segment in the segment set. That is,

> For every segment s_k of a segment set S, there corresponds a variable z_k from an ordering set Z. Any one-to-one function $f:f(Z)\rightarrow\{1,2,3,\ldots,|Z|\}$ specifies an order on the segment set such that if $z_i<z_k$, s_i precedes s_k in the program text for this ordering function.

In the original ordering of a program, $z_1=1$, $z_2=2$, $z_3=3$, and so on. Rearrangement changes the order of the segments and, thus, the values of Z.

> A rearrangement of a program is a one-to-one function f over the ordering set Z of the segments S of a program:
>
> $$f:f(Z)\rightarrow\{1,2,3,\ldots,|Z|\}$$
>
> such that for every $r_i=(s_i,s_j)$ in the program's rearrangement restriction set R, $z_i<z_j$. A rearrangement f of a program P is designated as P_f.

The purpose of rearrangement is minimization of the number of knots in a program. In order to count the knots in a program efficiently after reordering, a modification of the previous knot function is necessary. When rearranged, the line numbers of a program change relative to the statement content. In computing knots, a program should not be rebuilt from the segment set to determine the new line numbers of the entrances and exits of branches from which knots are determined. The solution is to specify the branches of a program in terms of the position of the corresponding entrances and exits within each segment. Although the segment ordering may change, the internal position of the statements will not. Therefore, each statement passing or receiving control flow is specified positionally by an offset

in lines from the beginning of the segment in which it is contained. That is, if a statement is on line 45 of a program and the third segment contains lines 40 through 48, the position of the statement is segment 3, offset 5.

Each statement in the knot graph is thus represented by a pair: (z_i,o_i), where z_i is the ordering variable for segment i and o_i is the offset in lines of this statement from the first statement in segment i. Each control flow edge is a 4-tuple: (z_i,o_i,z_j,o_j), where (z_i,o_i) is the position of the statement which jumps to a statement at position (z_j,o_j). When using this representation, although the value of z may change due to rearrangement functions, the value of each o is constant.

A knot is determined directly from this edge representation, bypassing a reconstruction of the program text for each ordering.

A <u>knot</u> exists between two knot-graph edges $e=(z_a,o_a,z_c,o_c)$ and $e'=(z_b,o_b,z_d,o_d)$ when,

$$s=z_a * |N| + o_a$$
$$t=z_b * |N| + o_b$$
$$u=z_c * |N| + o_c$$
$$v=z_d * |N| + o_d$$

if and only if

$$s<t<u<v \quad \text{or} \quad v<u<t<s$$
$$\text{or} \quad u<t<s<v \quad \text{or} \quad v<s<t<u$$
$$\text{or} \quad s<v<u<t \quad \text{or} \quad t<u<v<s$$
$$\text{or} \quad u<v<s<t \quad \text{or} \quad t<s<v<u.$$

Each z is multiplied by $|N|$ to insure that for any $z_a<z_b$, s<t. This follows because both o_a and o_b must be less than $|N|$.

As in section II, the knot count or knot number of a program is the number of pairs of control flow edges in the knot graph which knot. The knot count for some program P is represented as KNOT(P).

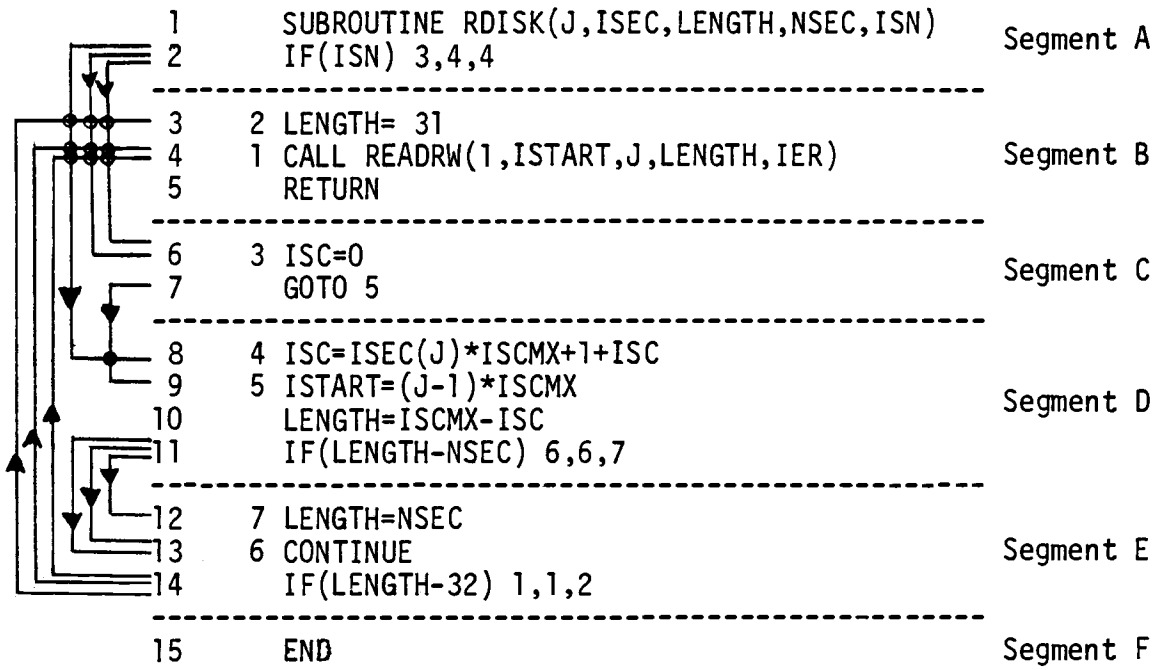
Finally, the rearrangement which produces the minmum knot count in a program can be defined.

A number k is the minimum knot number for some program P, if there is a rearrangement f_m for P such that for every other rearrangement f of P, $KNOT(P_{f_m}) \leq KNOT(P_f)$ and $KNOT(P_{f_m}) = k$.

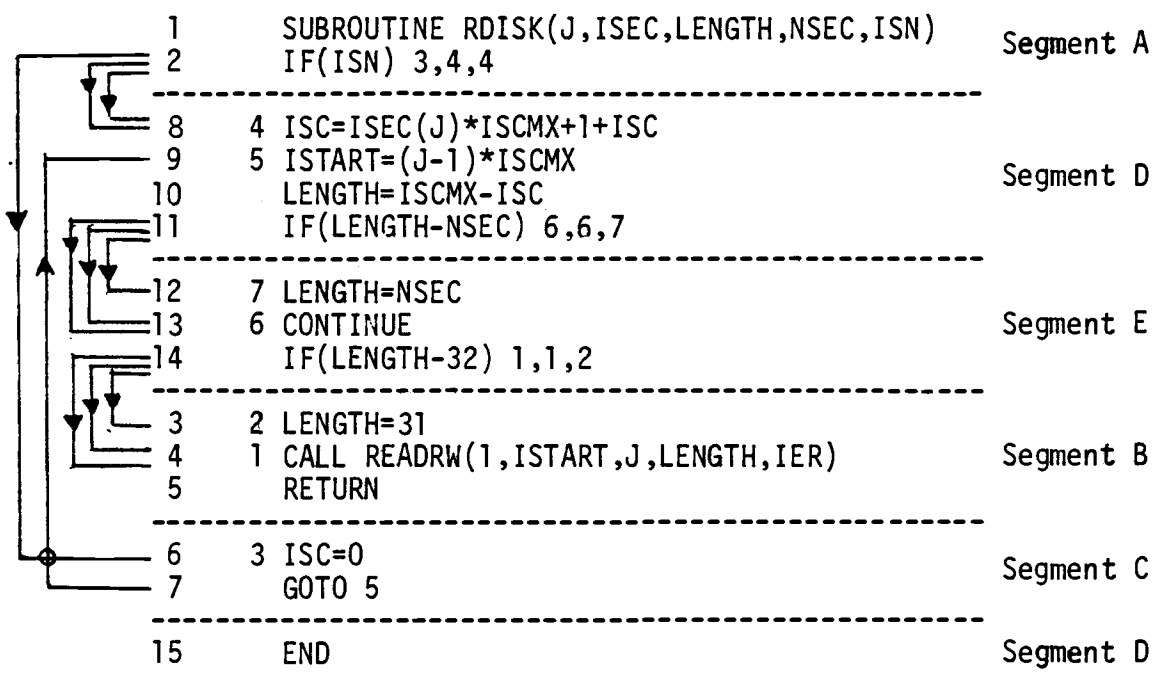
The rearrangement of a program which has a minimum knot number is a knot reordering minimum for that program.

Figure 12 is the reordering minimum result for a small program. The rearranged program is much clearer than the original. Although the knot-based reordering is demonstrably successful, the amount of computational effort to produce a less-tangled program code is questionable. Cook (3) suggests that rearrangement for minimum knot number is probably NP-complete, but leaves the question open. If knot reordering minimization is NP-complete, rearrangement of programs to produce minumum knot count would be prohibitive for any large programs. This paper will show a more general problem in rearrangement to be NP-complete on the size of the program to be rearranged for a knot minimum. The specific problem for knot graphs remains open, but fast algorithms are shown to approach the minimum solution for many programs.

The reordering algorithm can be converted to a decision problem for K, the number of knots in a program. The decision problem addresses the question of whether a particular solution exists, that is, whether



(I)



(II)

Figure 12: Program II is a minimum rearrangement of program I. The relative minimum of program I has a knot count greater than that of program II.

any rearrangement fuction can produce a specified number of knots for a program rearrangement.

The knot function, KNOT(P), previously specified, is altered slightly for definition of this problem. The argument of the KNOT moves from a program to a set of control flow edges found in a program. That is, KNOT(P) ≡ KNOT(E) where E is the set of edges in the knot graph of P. The two functions are obviously equivalent, except that the original has the added hardship of locating the control flow edges from program text. Both compute the knot number of a program in terms of a set of control flow edges and text basis.

MINIMAL KNOT REARRANGEMENT

Instance: A finite set B, a collection E of ordered pairs (b,b') from B, a collection R of ordered pairs (b,b') from B, and an integer K.

Problem: Is there a one-to-one function $f:f(B)\rightarrow\{1,2,3,\ldots, B\}$ such that KNOT(E) = K and, for all (b,b') in R, $f(b)<f(b')$?

The set B is the set of segments in a program, the set E, the edges, and the set R, the restrictions. The ordered pair of segments used for the edges does not account for statement offsets. The removal of the offsets is valid because programs with single-statement segments are at least as difficult to reorder to minimum knots as those where statement offset are required, as in multiple-statement segments. Clearly, any algorithm which can solve rearrangement for programs with multiple-statement segments and offsets can solve the single-statement, no offset programs as a special case.

The decision problem is obviously in NP. A generator can create all possible functions f and feed each to a non-deterministic machine. The machine rearranges a program from the input function f, checks rearrangement restriction satisfaction, and computes the knot number of the program. If the knot number matches K, a solution is found.

Generalization comes from the knot computation procedure developed above. The knot computation is over pairs of edges in the knot graph representation and therefore generalized to a 4-tuple of adjacent nodes of the edge pair under comparison for a knot. The set from which knots are computed is transformed from a set of statement pairs (nodes adjacent on an edge), where the number of members is $|E|$, to a set of statement 4-tuples (nodes adjacent of two edges), one for each unique edge pair from E, where the size of this set is $|E|(|E|-1)/2$. The generalized decision problem uses the 4-tuple representation. KNOT(D) is the knot count function for the set of 4-tuples. The function is equivalent to KNOT(E) except that KNOT(D) does not have to form all the pairs from the set E while computing the knot number of the set. All pairs are already formed in D.

GENERALIZED KNOT REARRANGEMENT

Instance: A finite set B, a collection D of ordered 4-tuples from B, a collection R of ordered pairs from B, and an integer K.

Problem: Is there a one-to-one function $f:f(B)\rightarrow\{1,2,3,\ldots, B\}$ such that KNOT(D) = K and for all (b,b') in R, $f(b)<f(b')$?

The generalized problem is in NP for the same reasons as the original problem. GENERALIZED KNOT REARRANGEMENT is shown NP-complete from a proven NP-complete problem. The problem chosen for the reduction is called the BETWEENNESS problem.(12,26)

BETWEENNESS

Instance: A finite set A and collection C of 3-tuples, (a,b,c) of distinct elements from A.

Problem: Is there a one-to-one function $f: A \rightarrow \{1,2,3,\ldots,|A|\}$ such that for all (a,b,c) in C,

$$f(a)<f(b)<f(c) \text{ or } f(a)>f(b)>f(c)?$$

BETWEENNESS was shown NP-complete from a reduction from SET-SPLITTING. (12) The reduction to GENERALIZED KNOT REARRANGEMENT uses a polynomial transformation of the sets A and C. For every 3-tuple in C, a 4-tuple is created for the set D.

$$(a,b,c) \rightarrow (a,b,c,z)$$

where z is not a member of A. The set B of GENERALIZED KNOT REARRANGEMENT is created from the set A and the variable z.

$$B = A \cup z$$

The set of restrictions, R, is built from z and all members of A. That is:

For all a in A, (a,z) is in R, and
for all (a,z) in R, a is in A.

The transformation to these sets is polynomial in the size of A. The number of 3-tuples in C cannot exceed $|A|^3$. Building the set D is $O(|A|^3)$. Building the set B requires $|A|+1$ operations and the set R, $|A|$ operations. The transformation is polynomial of order

$$O(|A|^3+2|A|+1)$$

operations.

To solve BETWEENNESS, GENERALIZED KNOT REARRANGEMENT is run on the transformed sets B, E, and R. The hit value of K is set to the size of C. The GENERALIZED KNOT REARRANGEMENT algorithm solves BETWEENNESS by finding (or not finding) a function f which has K knots. The value assigned z, f(z), must equal $|A|+1$ because of the restriction set. The only conditions for a knot when the last member of the 4-tuple is greater that any other member are:

$$f(a)<f(b)<f(c) \quad \text{or} \quad f(a)>f(b)>f(c),$$

where the 4-tuple is (a,b,c,z). If every 4-tuple produces a knot, the number of knots equals K. Any 4-tuple which does not knot reduces the number of knots to less than K: no 4-tuple can contribute more than one knot. The solution requires that for every (a,b,c) in C, b is between a and b. BETWEENNESS is polynomially reducible to GENERALIZED KNOT REARRANGEMENT. A theorem results

Theorem: GENERALIZED KNOT REARRANGEMENT is an NP-complete problem.

The proof that GENERALIZED KNOT REARRANGEMENT is NP-complete does not suffice for MINIMAL KNOT REARRANGEMENT. The condition for generalization from the rearrangement on the knot graph requires all pairs of edges to be represented as a single set of 4-tuples. GENERALIZED KNOT REARRANGEMENT and MINIMAL KNOT REARRANGEMENT are equivalent only if every algorithm which computes the problem the fastest requires comparison of all pairs of explicit edges to compute the knot content of an ordering of a program. Then,

$$\{(a,b),(c,d),(\ldots)\} \equiv$$
$$\{(a,b,c,d),(a,b,\ldots,\ldots),(c,d,\ldots,\ldots),(\ldots,\ldots,\ldots,\ldots)\}$$

in terms of any algorithm which reorders to minimum knots in the fastest time. Such a proof is beyond the scope of this paper and is left as an open problem.

One indication of the NP-completeness of reordering is the existence of relative minimums. In general, for some ordering of a program, exchange of any pair of elementary segments can produce a knot less than or equal to the knot count for any other exchange on the ordering. The result may not be the minimum knot count of the program. If for every program no such relative minimum knot count occurs, then some algorithm should exist to find a minimum knot number in polynomial time. An example of a program of this type is knot-graph represented in Figure 12. Minimization requires a shift of segment E between segment A and segment B. Every exchange towards that goal produces a greater knot count than in the original program.

Fast algorithms do exist for approximating the minimum knot number. Section 5 on empirical data contains results on the use of the relatively slow, relative minimization algorithm for finding the minimum knot number of programs. The algorithm was originally intended to find lower knot content orderings for statistical data on programs with many segments, but unexpectedly found the minimum knot number for a large number of the programs under test. Faster algorithms should have comparable success.

KNOTS AND STRUCTURED PROGRAMMING

Modular software construction and structured programming have shown a remarkable ability to enhance software quality.(1,5,6,27) In particular, the general areas of testability and maintainability have prospered. A metric which indicates the degree of modularity and structuredness of a program should significantly assist software development. Knots will be shown to provide such a metric.

Originally, structured programming arose from the concepts of software modularity.(7) A program module is a logically self-contained and discrete part of a larger program. Each module is a program or routine with a single entry point and single exit point. A complete, modular program consists of a collection of modules, which are functionally arrayed in a hierarchical fashion. That is, the complete, modular program has a main module which calls other modules which, in turn, call other modules. Structured programming extends this concept to the internal format of each module.

Structured programming attempts to create well-organized coding within programs. Its basis is the single entry-point, single exit-point rule implied by software modularity. In structured programming, the rule applies to sequences of statements within a program. The rule can be expressed in terms of the flow graph representation of a program as defined in Section 1.

The program block in a program is a sequence of statements with a single entrance point and single exit point. Two program blocks are

reducible to a single node in the flow graph when they can be combined (merging self-loops and parallel edges) without affecting the flow edges to and from any other basic blocks. If two program blocks can be reduced, they formed a modular sequence in the program. If the program flow graph can be iteratively reduced to a single node, the program is structured.

The flow graph definition captures the modularity of flow, but does not address the modularity of text, the other requirement of structuredness.(14) For a program to be modular in text, every sequence of statements which does not branch and is not branched into must be sequential in the program text. The definition of program structuredness is a combination of both the flow modularity and the text modularity.

> A program is structured when its flow graph is reducible to a single node and its text is modular.

Clearly the definition sets severe restrictions for structured programs. If structured implementations of algorithms are as powerful in every case as those which are unstructured, the need to bend those rules (and hence, the need for a metric to measure that variance) would not be justifiable. However, the need for this flexibility is pointed up by the current GOTO controversy.

In the process of definition of structured programming, many researchers have amended the original ideas of Dijkstra (7) to include the removal of GOTO's from programming practices.(2,21,31) The premise of the restriction is that GOTO's are the major source of control flow

tangle in programs. In FORTRAN, this is precisely the case. Explicit edges, other than DO-loops, are caused only by GOTO constructions. The restriction causes all control flow within a program to fall within the constructs of the programming language -- IF-THEN-ELSE, DO-WHILE, FOR, etc. Restriction to these conventions guarantees the program must be structured.

A more moderate approach was presented by Knuth.(20) While most programs work well under the GOTO restriction, he found restriction of some algorithms was both time and space inefficient. In some cases, removal of GOTO's actually increased the complexity of the program.

Many other arguments have been presented on both sides of the question, but some middle ground can be found.(18) Techniques which restrain rather that abolish GOTO's to produce "slightly" unstructured programs is one solution.

Among other advantages, such restraining techniques allow the IF-THEN-ELSE construct to appear in FORTRAN programs. Without GOTO's, FORTRAN is limited to IF-THEN structures. Use of a GOTO terminates the THEN part of the structure with a branch around the section performing the ELSE. The structure may be formed by using two or three GOTO's.(24,29,30,33) Figure 13 shows these two forms. The non-GOTO structure for this code is:

```
IF (A) CALL THEN ( ... )
IF (.NOT.A) CALL ELSE ( ... )
```

Such code, however, makes it difficult to understand the purpose and function of the IF-THEN-ELSE. The processing of the CALL also slows

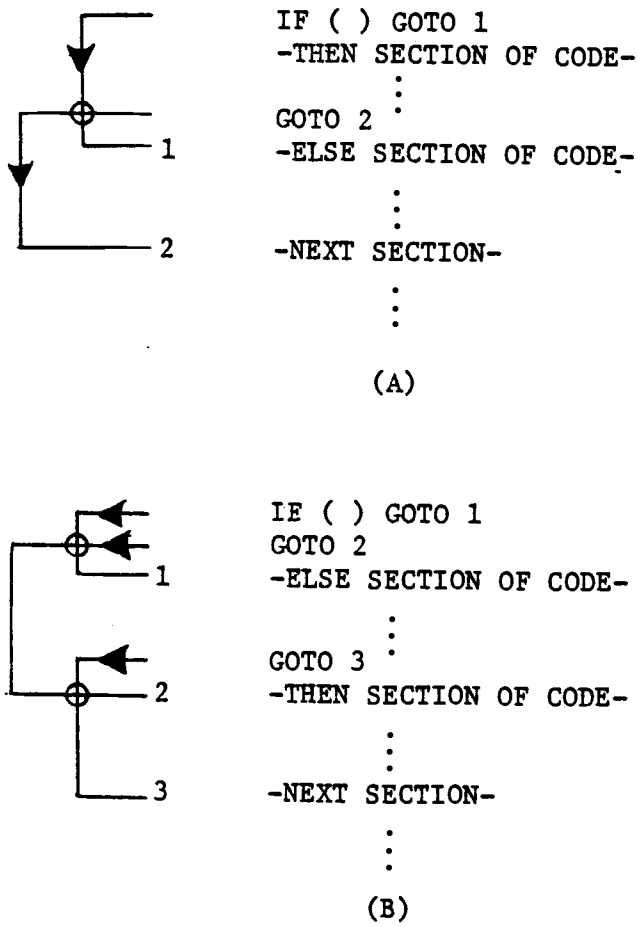


Figure 13: Two formats for FORTRAN IF-THEN-ELSE. Type A has one knot and type B has two knots.

the procedure.(20,31)

One characteristic immediately apparent in the above code is that the unconditional GOTO's cause a knot in the structured IF-THEN-ELSE. The knot number of the first type (A) is one, and the second type (B) is two. If the above sections are defined as the structured representations of a FORTRAN IF-THEN-ELSE, then an IF-THEN-ELSE causes a knot. The result applies to other languages as well, if the restriction of conversion of single-line, multiple-statement, control functions to multiple-line, single statement format is specified. That is,

```
if ( ) then ( ) else;     ->   if ( )
                                  then ( )
                                  else ( );
```

The implied edges of this structure, when control flow is drawn, show a knot.

The examples demonstrate that although a program with zero knots must be structured, a structured program does not necessarily have zero knots. The knot relationship as presented by Woodward et al. (30) was found in the flow reduction of a program. McCabe (23) showed that a program was structured if its flow graph could be reduced to a single node. The corresponding result of (30) is that every structured program has a labeled flow graph representation which is reducible to a flow graph with zero knots. One drawback is that the labeled flow graph must be reduced to determine structuredness. Another drawback is that the labeled flow graph with its innate deficiencies is used.

The problem is simplified if the structuredness of a program may

be determined directly from the knots in the unreduced knot graph. The solution is to further specify the knot by type, using some characteristic of each knot in the program indicative of the structuredness of the edges which caused it. If a program contains unstructured knots, the program is unstructured. The number of unstructured knots indicates the degree of unstructuredness of the program. These knots are also the knots remaining when a reduction is applied to the knot graph.

The most obvious knot formation that breaches modularity is one which violates a DO-loop structure in FORTRAN. A DO-loop is defined as any backwards branch in the text. Backwardness is based on the text rather than the program control flow, even though the branch may not be a functional iteration. A structured text should reflect the structure of program operation. If a backwards branch occurs in the text which is not an iteration loop, the text is not reflecting program operation and violates the text modularity condition of the definition. Figure 14 shows a program section with this lack of correspondence. Any program containing a backwards branch which is not a loop is unstructured. All backwards branches in a structured program must be loops.

Often an unstructured program passes control to or from the interior of a loop. In its knot graph, an unstructured program with this type of violation must have a knot on the loop edge. Figure 15 shows the possible knot subgraphs for this type of unstructuredness. The knot characteristic for a loop knot is found directly from the edge representation involved. A knot is <u>positive</u> if neither edge is a loop (backwards branch). Otherwise a knot is <u>negative</u>.

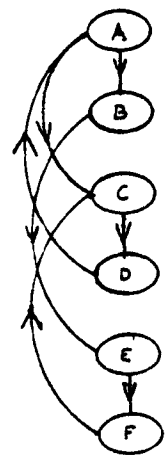


Figure 14: Textually non-modular statement in knot graph.

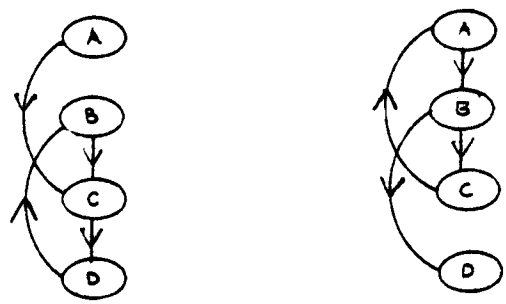


Figure 15: Exit from and entrance to knot graph loop.

> Given e=(a,c) and e'=(b,d) where a, b, c, and d are line numbers of a program, and e and e' are control flow edges, (a,c) implies edge e exits statement a and enters at statement c. If e and e' knot, the knot is positive if and only if a<b<c<d. The knot is negative otherwise.

Clearly every entrance (exit) to (from) the inside of a functional loop will be detected by a negative knot. The edge of a loop, e=(a,c) must have a>c. The same is true for textual backward branches. Non-iterative backwards branches occur when a section of the program is displaced from the text module of code which should properly contain it. (Figure 16)

In Figure 16, case II, a negative knot is automatically found, so only cases I, III, and IV need examination. In each of these cases, the segment of block B must be entered and exited from or to some other part of the program. If B is entered from any place other than A, a negative knot occurs. If B is entered from A, the exit may be to A, C, or some other segment. If B exits to A or B, there are no violations to program text structuredness: cases I, III, and IV are well-defined, structured modules. If B exits to some other section, then a negative knot occurs, and the program is unstructured. The result is equivalent if C and D are at lesser line numbers than A and B. Thus, structured programs cannot contain negative knots.

The result applies if there is a number of segments in the program, or a single segment. Unstructuredness conditions when positive knots occur in a program are not as independent. The framework of the IF-THEN-ELSE is FORTRAN (or any other language) imposes a positive

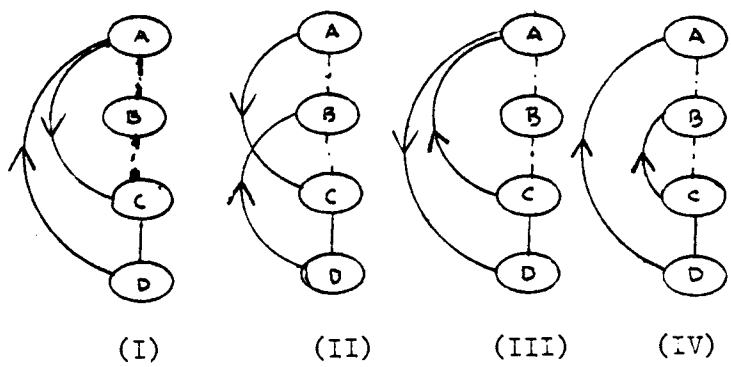


Figure 16: Backwards branches in knot graphs.

knot in the program. Nested IF-THEN-ELSE's cause multiple positive knots. The IF-THEN-ELSE is characterized by the segmentation which the construction causes. The THEN, the ELSE, or both sections are contained in different elementary segments. The observation implies that the unstructuredness of a program is related to the containment of edges causing knots by elementary segments.

The edge containment of knots is divided into two types. If the edges causing a knot are contained in at most two segments, the knot is internal. If the edges are contained in at least three segments, the knot is external.

> Given e=(a,b) and e'=(c,d) where e and e' are control flow edges in a program and a, b, c, and d are the elementary segments upon which e and e' are incident, (a,b) implies an edge exits segment a and enters segment b. If e and e' knot, the knot is internal if a=b=c or a=b=d or a=c=d or b=c=d. The knot is external if at most any two members of the set {a,b,c,d} are equal.

The definition relates a knot to McCabe's (23) unstructuredness results. McCabe showed that the existence of a particular type of subgraph of the program flow graph, without unconditional GOTO's, was sufficient and necessary for unstructuredness in the program. In Figure 17, the graphs represent entrance to a loop, exit from a loop, entrance to a decision, and exit from a decision. Knots have previously been shown to detect subgraphs of types II, III, and IV. The remaining type, I, is a positive knot. The restriction of no unconditional GOTO's constrains the programs under consideration to a single elementary segment for the whole program. However, the subgraph result must also hold for each elementary segment in a program with multiple segments.

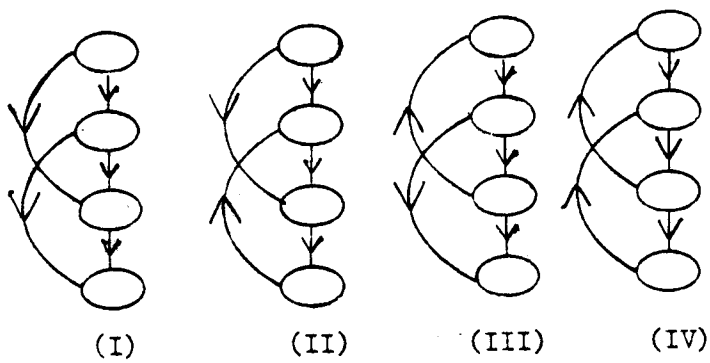


Figure 17: The four unstructured subgraphs from (23).

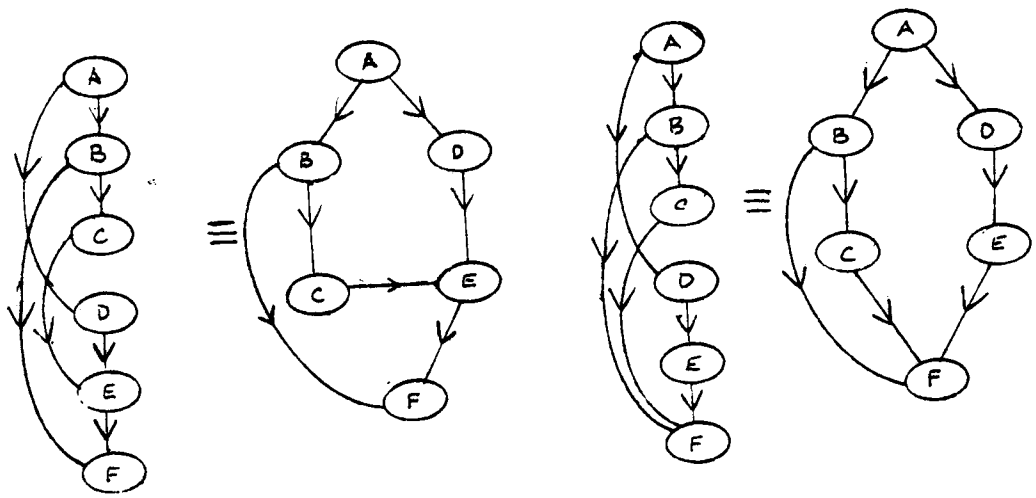
The GOTO constraint holds within each elementary segment. Internal, positive knots clearly indicate unstructuredness.

The type of knot remaining, positive and external, is not sufficient to determine unstructuredness or prove structuredness. That is, a program with positive, external knots may or may not be unstructured in a FORTRAN program without language constraints. An example is the two knot subgraphs shown in Figure 18.

The subgraphs in Figure 18-1 and Figure 18-2 each contain two positive, external knots. However, the subgraph in 1 is unstructured and in 2 is structured. The knots in both graphs are identical in every way except for two entrance nodes, e and f. Knot graph specification of two edges is insufficient to differentiate the two cases. Absolute unstructuredness must still rely on some reduction of the program, although a very useful theorem results form the above discussion.

Obviously knots are partitioned into positive and negative knots, and these sets are further partitioned into internal and external knots. The partitions containing negative-internal and negative-external knots are empty if a program is structured. The partition containing positive-internal knots is empty if the program is structured. The partition containing positive-external knots is the only one to which membership does not imply unstructuredness. Therefore, all knots, if any, in a program must be positive-external if the program is structured.

Theorem: A structured program may contain only positive-external knots.



(1) -- unstructured (2) -- structured

Figure 18: Structured and unstructured knot graphs with the same positive knot type.

Although the result limits the ability for a knot comparison between edges to determine structured programs uniquely, the result has more potential in application than any previous metric. The cyclomatic number, for instance, indicates nothing about the structuredness of a program until the flow graph to the program has been reduced. Line counts, operator counts, GOTO counts, branch counts, or any combination thereof is even weaker. The association of the knot with structuredness establishes that if the technique is a major controlling factor of complexity, the knot provides an equally effective measure of that control.

EMPIRICAL RESULTS

I implemented an automatic system to examine the knot concepts over a functioning set of programs. I sought several results from the experiment. The first was a confirmation of the results of a similar experiment (30) on a larger and less constrained set of programs. Secondly, I was looking for the degree to which programmers write code which cannot be rearranged to cause fewer knots. My third objective was to show that although relative minimums in the knot count of rearranged programs exist, segment exchange quickly creates a reordering of a program in which the knot count approaches the minimum knot count for any rearrangement of the program. The fourth aspect of the experiment concerned the ease of implementation of the knot measure. I needed some indication of the developmental and functional overhead in automation of the knot measure.

In the first stage of the experiment, I built a routine to compute the following characteristics of a FORTRAN program.

1. Program length -- the number of statements in a program, not including non-functional statements.
2. Segments -- the number of elementary segments as specified in Section 3 in a program.
3. Cyclomatic number -- The number of linearly independent circuits in the flow graph of a program.
4. Knot count -- the number of knots found in the original ordering of the statements of a program.
5. Relative-minimum knot count -- the least knot count for the rearrangement of a program using segment exchange. The knot

count is the lowest when no segment exchange on that ordering produces an ordering of the segments with a lesser knot count.

6. Relative-minimum exchanges -- the number of exchanges necessary to find a relative minimum.
7. Absolute-minimum knot count -- the minimum number of knots found for every rearrangement of a program.
8. Absolute-maximum knot count -- the maximum number of knots found for any rearrangement of a program.
9. Average knot count -- the average knot count over all rearrangements of a program.
10. Reorderings -- the number of possible rearrangements of a program.

In the second stage of the experiment, I computed correlations between each of the above characteristics over all the programs tested.

The software for computing these values consisted of a two-pass, program flow analyzer, a knot calculator, a segment order exchange routine for relative minimization, and a permutation generator for creating all rearrangements of a program. The program analyzer converted a FORTRAN program to edge representations of the explicit branches in the text. The routine stored the segment number and state-offset on exit and segment number and statement offset on entrance for each explicit branch in a program. The first pass partitioned a program into elementary segments and stored the segment/offset values for each labeled (entry) statement in a table. The second pass found all explicit branches and stored the segment/offset value of the branching statement with the jump label value for the table. Thus, the edge set for a program was stored in the form suggested in Section 3.

The knot calculation routine computed the number of knots in the

edge set for any particular ordering of the segments. The routine maintained a one-dimensional array of length equal to the number of segments in a program. The segment values in the edge set were indices to this array. In this way, a program reordering consisted of placing integers, from one to the number of segments, in some sequence in the array. The knot calculation routine compared every pair of edges in the edge set, using the segment value to index the array for the current ordering number of that segment. Initially, each array value was set equal to its index. The knot calculation on that sequence gave the knot count of the program of the original ordering of the segments.

After the edge set of the program had been compiled and the original knot count computed, the system performed two types of reordering of the elementary segments. For determination of a relative-minimum knot count, two segments at a time were exchanged to create a new ordering of the program. If the new ordering did not produce a lesser knot count, a new pair of segments were exchanged relative to the previous ordering. If no exchange of segments on that ordering produced a lesser knot count, the count was saved as the relative-minimum knot count and the number of exchanges to that point was stored as the relative-minimum exchanges. If a new ordering had a lesser knot count upon segment exchange, the segment exchanges continued with that ordering as its base.

If there were not more than eight segments in a program, the system computed the knot count for every rearrangement of the program. The least and greatest knot counts found were the absolute-minimum knot count and absolute-maximum knot count for the program. The

system did not check for DO-loop restrictions; none of the programs tested required them. The sum of the absolute-maximum and absolute-minimum knot counts divided by two gave the mid-range or average knot count for a program. The reorderings of a program were equal to the factorial of the number of segments less two -- the initial and final segments which could not be rearranged.

Once the characteristic values of the set of programs had been computed and stored, a routine computed the correlations between every pair of characteristics. The correlation formula implemented was:

$$r = \frac{N \Sigma x y - (\Sigma x \Sigma y)}{[N \Sigma (x^2) - (\Sigma x)^2]^{\frac{1}{2}} [N \Sigma (y^2) - (\Sigma y)^2]^{\frac{1}{2}}} \quad (32)$$

The system collected and correlated these values for 155 programs used on the Oregon State University, School of Oceanography, Geophysics Group computer system. All routines were in the category of scientific, numerical software and written in FORTRAN. Most programs were small; on the average, each had less than one hundred lines of code. Large array processing, serial data processing, plotting, and statistical routines were the most common in the sample. The spread in knowledge and experience of the programmers was quite large. The programmers varied from inexperienced, non-computer science graduate students, to experienced researchers, to expert programmers. The sample represented a wide range of scientific software and programmmer skill, without prefiltration by software construction controls.

Table 1 is a compilation of the program data for these routines,

sorted with respect to program length. The blank entries under the last four columns indicate that the program had too many segments for practical computation of these values. The upper-bound for these computations was set at eight elementary segments.

The correlations on this data sample are shown in Table 2. Table 3 shows the correlation probability for a sample of this size and is the correlation base of the sample. That is, there is a 1 percent probability that a sample base of this size would show a correlation of greater than .259 randomly. The program number in the table is a number arbitrarily assigned to each program as it was processed. The correlations with other variables reflects the pseudo-randomness with which the programs were ordered for analysis. The very low correlations with all other variables provides an additional basis for judging the degree of correlation between other variables.

The first result of the experiment shows that the correlation of the cyclomatic number with the knot count (.569) is approximately the same as that found in (30), (.600). Moderate correlation may imply more about the dependence of knots on branching than a direct relationship between the two metrics. The high correlation (.843) of cyclomatic number with program length also supports the results of (30), (.98). The lesser correlation between knots and program length (.508) further implies a difference in the two measures, at least for the programs tested.

The second result of the experiment shows a higher correlation (.914) of the absolute-maximum knot count with the original knot count than the absolute-minimum knot count with the original knot count (.785).

Although both correlations are high, the difference is meaningful because in the majority of the programs, no reordering was done. The data list shows that where absolute-minimum and absolute-maximum knot counts were found, a smaller knot number was found more often than not. When relative minimization supplied the only knot count minimum data, a lower value was also found. At least for this sample, programs contain more knots than necessary for the code used, and, in general, greater than the mid-range of the knot counts possible for any ordering of the code.

The third result derives from the correlation between the relative minimum knot count and the absolute-minimum knot count. In only 12 of the 80 programs where rearrangement was practical or possible, the relative minimization did not find the absolute-minimum knot count. In the five programs with greater than ten knots which could not be practically reordered, only one relative-minimization did not find a program ordering with significantly fewer knots than the original. Although finding the ordering of a program with the absolute-minimum knot number is difficult, relative-minimization succeeds in significantly reducing the knot count. In most cases, the reduction is also to the absolute-minimum knot number.

The efficiency with which knots may be implemented was demonstrated by the development of the experiment. The software required about two days to write and debug, and the 155 programs in the data sample took about four hours, wall-clock time, to test. The basic knot computation for a program required approximately 1 percent of the average compile time for the program. Reordering, however, required larger amounts of

time and computational effort. In general, a knot measure could be added effectively to a software development system with minimal development and functional overhead.

Table 1: Program data from knot testing routine.

LEN -- Program length
SEG -- Number of segments
CYC -- Cyclomatic number
KNT -- Knot count
RMK -- Relative minimum knots
RX -- Relative minimum exchanges
AMX -- Absolute maximum knots
AMN -- Absolute minimum knots
AVK -- Average knots
PRM -- Segment reorderings

LEN	SEG	CYC	KNT	RMK	RX	AMX	AMN	AVK	PRM
340	25	46	35	35	253				
172	6	23	44	41	8	46	41	43	24
168	13	22	26	16	227				
140	9	36	16	16	21				
130	7	13	11	10	11	23	9	16	120
129	3	24	21	21	0	21	21	21	1
120	2	19	3	3	0	3	3	3	1
115	7	17	37	23	19	55	22	39	120
115	5	23	6	6	3	7	6	7	6
114	12	20	5	4	54				
114	14	25	5	5	66				
112	5	23	6	6	3	7	6	7	6
111	2	15	2	2	0	2	2	2	1
110	6	26	118	28	18	125	28	77	24
110	10	13	8	6	75				
109	8	16	22	17	30	26	17	22	120
108	5	23	6	6	3	7	6	7	6
108	5	17	12	10	4	12	10	11	6
104	1	3	0	0	0	0	0	0	1
103	7	15	5	5	10	19	3	11	120
97	3	12	2	2	0	2	2	2	1
95	4	16	16	16	1	16	16	16	2
94	19	33	8	2	330				
88	10	25	1	1	28				
86	2	10	2	2	0	2	2	2	1
86	8	14	10	7	21	42	7	25	720
84	3	8	6	6	0	6	6	6	1
81	10	16	6	6	28				
77	4	9	4	4	1	12	4	8	2
76	11	13	0	0	36				
75	13	26	17	3	239				
74	5	11	10	10	3	18	10	14	6
74	2	4	0	0	0	0	0	0	1
72	14	23	32	19	245				
72	4	10	4	4	1	10	4	7	2
72	4	7	0	0	1	0	0	0	2
70	5	6	2	1	4	2	1	2	6
67	7	17	24	22	11	66	22	44	120
65	3	3	0	0	0	0	0	0	1
63	2	13	4	4	0	4	4	4	1

Table 1: Program data from knot testing routine (cont.)

LEN	SEG	CYC	KNT	RMK	RX	AMX	AMN	AVK	PRM
60	4	13	12	11	2	12	11	12	2
60	5	7	3	2	4	4	2	3	6
57	3	7	0	0	0	0	0	0	1
54	2	8	0	0	0	0	0	0	1
53	12	15	7	4	104				
53	8	11	15	5	35	39	4	22	720
52	8	11	15	5	35	39	4	22	720
51	1	4	2	2	0	2	2	2	1
50	6	10	3	0	13	3	0	2	24
50	2	9	0	0	0	0	0	0	1
50	3	6	4	4	0	4	4	4	1
49	1	7	0	0	0	0	0	0	1
49	3	6	4	4	0	4	4	4	1
48	1	9	0	0	0	0	0	0	1
48	3	7	1	1	0	1	1	1	1
48	3	4	1	1	0	1	1	1	1
47	2	5	0	0	0	0	0	0	1
47	3	3	0	0	0	0	0	0	1
46	7	16	3	3	10	11	3	7	120
46	2	8	6	6	0	6	6	6	1
45	2	9	0	0	0	0	0	0	1
45	4	3	0	0	1	0	0	0	2
44	5	12	2	2	3	7	2	5	6
43	2	8	0	0	0	0	0	0	1
43	8	7	4	2	28	24	0	12	720
42	8	7	4	2	28	24	0	12	720
42	2	3	0	0	0	0	0	0	1
41	5	9	1	1	3	1	1	1	6
41	3	8	6	6	0	6	6	6	1
40	2	4	0	0	0	0	0	0	1
40	4	4	1	0	2	1	0	1	2
39	2	14	0	0	0	0	0	0	1
38	6	11	4	3	12	8	3	6	24
38	2	9	0	0	0	0	0	0	1
38	3	8	3	3	0	3	3	3	1
37	6	9	12	1	13	15	0	8	24
36	6	9	12	1	13	15	0	8	24
36	2	2	0	0	0	0	0	0	1
36	2	2	0	0	0	0	0	0	1
34	2	3	0	0	0	0	0	0	1
33	3	5	4	4	0	4	4	4	1
32	7	14	14	4	16	24	3	14	120
32	6	9	9	1	13	13	0	7	24
32	5	7	0	0	3	0	0	0	6
32	3	4	2	2	0	2	2	2	1

Table 1: Program data from knot testing routine (cont.)

LEN	SEG	CYC	KNT	RMK	RX	AMX	AMN	AVK	PRM
31	3	4	2	2	0	2	2	2	1
31	2	3	0	0	0	0	0	0	1
30	2	9	21	21	0	21	21	21	1
30	3	5	0	0	0	0	0	0	1
30	6	5	4	2	12	13	0	7	24
29	1	8	0	0	0	0	0	0	1
28	2	3	1	1	0	1	1	1	1
27	2	10	0	0	0	0	0	0	1
27	2	4	0	0	0	0	0	0	1
26	2	3	0	0	0	0	0	0	1
26	1	2	0	0	0	0	0	0	1
25	6	7	4	0	7	4	0	2	24
25	2	2	0	0	0	0	0	0	1
24	2	5	0	0	0	0	0	0	1
24	3	2	0	0	0	0	0	0	1
24	2	2	0	0	0	0	0	0	1
23	4	5	1	0	2	1	0	1	2
23	4	5	4	3	2	4	3	4	2
23	2	3	0	0	0	0	0	0	1
23	2	2	0	0	0	0	0	0	0
22	5	6	2	2	3	2	2	2	6
22	3	5	1	1	0	1	1	1	1
22	2	3	1	1	0	1	1	1	1
22	1	3	0	0	0	0	0	0	1
22	3	3	1	1	0	1	1	1	1
21	2	3	0	0	0	0	0	0	1
20	2	5	1	1	0	1	1	1	1
20	2	3	0	0	0	0	0	0	1
20	2	1	0	0	0	0	0	0	1
19	4	4	2	0	2	2	0	1	2
18	6	7	2	2	6	13	0	7	24
18	2	1	0	0	0	0	0	0	1
17	4	6	0	0	1	0	0	0	2
17	4	3	2	2	1	3	2	3	2
17	1	1	0	0	0	0	0	0	1
17	1	1	0	0	0	0	0	0	1
15	2	5	1	1	0	1	1	1	1
15	2	3	0	0	0	0	0	0	1
14	4	5	0	0	1	4	0	2	2
14	2	4	1	1	0	1	1	1	1
13	5	7	9	0	9	9	0	5	6
13	2	6	0	0	0	0	0	0	1
13	3	4	1	1	0	1	1	1	1
13	2	4	1	1	0	1	1	1	1
13	2	3	0	0	0	0	0	0	1

Table 1: Program data from knot testing routine (cont.)

LEN	SEG	CYC	KNT	RMK	RX	AMX	AMN	AVK	PRM
13	2	3	0	0	0	0	0	0	1
13	2	1	0	0	0	0	0	0	1
13	2	1	0	0	0	0	0	0	1
12	4	4	2	2	1	2	2	2	2
12	3	3	0	0	0	0	0	0	1
12	2	3	0	0	0	0	0	0	1
12	2	2	0	0	0	0	0	0	1
12	2	2	0	0	0	0	0	0	1
12	2	2	0	0	0	0	0	0	1
12	2	2	0	0	0	0	0	0	1
12	2	2	0	0	0	0	0	0	1
12	2	1	0	0	0	0	0	0	1
12	2	1	0	0	0	0	0	0	1
11	2	6	0	0	0	0	0	0	1
11	2	2	0	0	0	0	0	0	1
10	2	3	0	0	0	0	0	0	1
10	2	1	0	0	0	0	0	0	1
9	2	4	0	0	0	0	0	0	1
9	2	1	0	0	0	0	0	0	1
8	2	2	0	0	0	0	0	0	1
7	2	3	0	0	0	0	0	0	1
7	2	2	0	0	0	0	0	0	1
6	2	1	0	0	0	0	0	0	1
5	2	1	0	0	0	0	0	0	1

Table 2: Program data correlations from knot test routine.

	PR#	LEN	SEG	CYC	KNT	RMK	RX	AMX	AMN	PRM	BW	AVK	
PR#		.215	.229	.208	.078	.230	.113	.113	.190	.153	.036	.142	Program number
LEN	.215		.661	.843	.508	.719	.487	.493	.655	.193	.178	.567	Program length
SEG	.229	.661		.778	.416	.493	.851	.630	.393	.617	.241	.587	Number of segments
CYC	.208	.843	.778		.569	.686	.613	.626	.683	.219	.268	.674	Cyclomatic number
KNT	.018	.508	.416	.569		.800	.325	.914	.785	.186	.734	.925	Knot count
RMK	.230	.719	.493	.686	.800		.352	.772	.997	.191	.408	.882	Relative minimum knots
RX	.113	.487	.851	.613	.325	.352		.664	.311	.864	.024	.589	Relative minimum exchanges
AMX	.113	.493	.630	.626	.914	.772	.664		.749	.439	.939	.980	Absolute maximum knots
AMN	.190	.655	.393	.683	.785	.997	.311	.749		.155	.475	.866	Absolute minimum knots
PRM	.153	.193	.617	.219	.186	.191	.864	.439	.155		.503	.376	Segment reorderings
BW	.036	.178	.241	.268	.734	.408	.024	.939	.475	.503		.851	Knot range in reordering
AVK	.142	.567	.587	.674	.925	.882	.589	.980	.866	.376	.851		Knot average in reordering

Table 3: Correlation probabilities (24) for 155 samples.

Correlation	Probability
.168	.100
.199	.050
.235	.020
.259	.010
.327	.001

CONCLUSIONS

To summarize, I would like to put the findings of this thesis in the perspective of the goal stated in the intoduction: implementation of the knot metric as a software development tool.

A brief look at the characteristics of the knot indicates that the metric is closely related to both program attributes associated with program complexity: program text structure and control flow. The representation of those attributes with a knot graph allows a fast algorithm to enumerate the knots in any program. The optimal algorithm of that process was found to be $O(\varepsilon')$ in space and $O(\varepsilon'^2)$ in time for the number ε' of edges in a knot graph with neither text-adjacent or text-bounding end-points.

Using the knots of a program to reduce its complexity is a valuable tool, but, theoretically, that process of minimal rearrangement is too difficult in computational complexity to implement. Approximating the minimum rearrangement, however, is fast, and for the most part, reaches the ordering which has the minimum knot number. Knots, then, can be used effectively to refashion programs to less complex textual structures in the practical domain.

Refining the knot to special types depending on the direction and containment of edges in a segmented knot graph produces a tool for measuring the structuredness of programs. Structuredness is closely tied to complexity; therefore, the knot characteristics will provide a direct control on the creation of almost-structured, yet less complex,

programs.

The empirical data which I accumulated justifies the knot metric from yet another standpoint. The measure is relatively independent of previous complexity measures and also assesses a different aspect of that complexity. In addition, it is relatively easy to implement and operates with low overhead.

This thesis primarily concerns FORTRAN programs and FORTRAN programming, but is extendable to other programming languages, even those without a GOTO type instruction. Any IF-THEN-ELSE causes a knot, so the knot measure in other languages detects the density of IF-THEN-ELSE constructions. Because such constructions often are associated with program complexity, knots are applicable to other languages than FORTRAN.

In general, the knot metric appears to be a practical, multi-purpose software complexity measure and tool.

REFERENCES

1. Baker, F. T., "Chief Programmer Team Management of Programming," *IBM Systems Journal,* January 1972, pp. 59-73.

2. Clark, R. L., "A Linguistic Contribution to GOTO-less Programming," *Datamation,* V. 19, No. 12, December 1973, pp. 62-88.

3. Cook, C. R., "Graph Theoretic Program Complexity Measures," *Department of Computer Science Technical Report, Oregon State University,* May 1979.

4. Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., Love, T., "Measuring the Psycological Complexity of Software Maintainance with the Halstead and McCabe Metrics," *IEEE Transactions on Software Engineering,* V. SE-5, No. 2, March 1979, pp. 96-104.

5. Curtis, B., Sheppard, S. B., Milliman, P., "Third Time Charm: Stranger Prediction of Programmer Performance by Software Complexity Metrics," *Proceedings of the 4th International Conference on Software Engineering,* Munich, September 1979, pp. 356-360.

6. Dijkstra, E. W., "The structure of 'THE' Multiprogramming System," *Communications of the ACM,* V. 11, No. 5, May 1968, pp. 341-346.

7. Dijkstra, E. W., "Programming Considered as a Human Activity," *Proceedings, IAP Congress 65,* North Holland, Amsterdam, 1965, pp. 213-217.

8. Dijkstra, E. E., "GOTO Statement Considered Harmful," *Communications of the ACM,* V. 11, No. 3, March 1968, pp. 147-148.

9. Elshoff, J. L., Marcotty, M., "On the Use of the Cyclomatic Number to Measure Program Complexity," *ACM SIGPLAN Notices,* December 1978, pp. 29-39.

10. Fitzsimmons, A. B., Love, L. T., "A Review and Evaluation of Software Science," *ACM Computing Surveys,* V. 10, 1978, pp. 3-18.

11. Gannon, J. D., Horning, J. J., "Language Design for Programming Reliability," *IEEE Transactions on Software Engineering,* V. SE-1, No. 2, June 1975, pp. 179-191.

12. Garey, M. R., Johnson, D. S., *Computers and Intractability,* Freeman, San Francisco, 1979.

13. Gilb, T., *Software Metrics,* Winthrop, Cambridge, Mass, 1977.

14. Gordon, R. D., "Measuring Improvements in Program Clarity," *IEEE Transactions on Software Engineering,* V. SE-5, No. 2, January 1979, pp. 79-90.

15. Gordon, R. D., "A Quantitative Justification for a Measure of Program Clarity," *IEEE Transactions on Software Engineering,* V. SE-5, No. 2, March 1979, pp. 121-128.

16. Halstead, M. H., "Toward a Theoretical Basis for Estimating Programming Effort," *Proceedings of the ACM, 1975,* ACM, New York, 1975, pp. 222-224.

17. Hansen, W. J., "Measurement of Program Complexity by the Pair," *ACM SIGPLAN Notices,* March 1978, pp. 29-33.

18. Hopkins, M. E., "A Case for the GOTO," *Proceedings of the 25th ACM National Conference,* V. 2, 1972, pp. 787-790.

19. Kerrigan, B. W., Plauger, P. J., *The Elements of Programming Style,* McGraw-Hill, New York, NY, 1978.

20. Knuth, D. E., "Structured Programming with GOTO Statements," *Current Trends in Programming Methodologies, Volume I,* Prentice Hall, Englewood Cliff, N. J., 1977, pp. 140-194.

21. Leavenworth, B. M., "Programming With(out) the GOTO," *Proceedings of the ACM 25th National Conference,* V. 2, 1972, pp. 782-786.

22. Love, T.,"An Experimental Investigation of the Effect of Program Structure on Program Understanding," *ACM SIGPLAN Notices, Language Design for Reliable Software,* V. 12, March 1977, pp. 3-18.

23. McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering,* V. SE-2, No. 4, December 1976, pp. 308-320.

24. Merchant, M. J., *Applied FORTRAN Programming with Standard FORTRAN, WATFOR, WATFIV, and Structured WATFIV,* Wadsworth, Belmont, Cal., 1977.

25. Myers, G. J., "An Extension to the Cyclomatic Number," *ACM SIGPLAN Notices,* October 1977, pp. 61-64.

26. Oputiny', J., "Total Ordering Problem," Unpublished manuscript, 1978.

27. Schneidewind, N. G., Hoffman, J. M., "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions on Software Engineering,* V. SE-5, No. 3, May 1979, pp. 276-286.

28. Tarjan, R. E., Yao, A. C., "Storing a Sparse Table," *Communications of the ACM*, V. 22, No. 11, November 1979, pp. 606-611.

29. Tenny, T., "Structured Programming in FORTRAN," *Datamation*, V. 20, July 1974, pp. 110-115.

30. Woodward, M. R., Hennel, M. A., Hedley, D., "A Measure of Control Flow Complexity in Program Text," *IEEE Transactions on Software Engineering*, V. SE-5, No. 1, January 1979, pp. 45-50.

31. Wulf, W. A., "A Case Against the GOTO," *Proceedings of the 25th ACM National Conference*, V. 2, 1972, pp. 791-797.

32. Young, H. D., *Statistical Treatment of Experimental Data*, McGraw-Hill, N.Y., 1962.

33. Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, Englewood Cliffs, N.J., 1975.

34. Zelkowitz, M. V., "Automatic Program Analysis and Design," *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco, 1977, pp. 158-164.