COMPLEXITY MEASURES FOR PROGRAMMING LANGUAGES

Technical Memorandum 17

(This report was reproduced from an M.S. Thesis, MIT,
Dept. of Electrical Engineering, September 1971.)

Leonard I. Goodman

September 1971

PROJECT MAC

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

ACKNOWLEDGEMENTS

I wish to thank Professor John Donovan, the supervisor of this thesis, for his encouragement, enthusiasm, and guidance during the research and preparation of this report. His help is deeply appreciated.

I acknowledge fellow graduate student Jerry Johnson for the many discussions we had during the early formulation of this work, and Cathy Doyle for her typing of this thesis report.

Finally, I thank my wife Mindi for her patience and encouragement during my graduate study.

Work reported herein was supported in part by Project MAC, an M. I. T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction of this document, in whole or in part, is permitted for any purpose of the United States government.

COMPLEXITY MEASURES FOR PROGRAMMING LANGUAGES*

Leonard I. Goodman

Abstract

A theory of complexity is developed for algorithms implemented in typical programming languages. The complexity of a program may be interpreted in many different ways; a method for measuring a specific type of complexity is a complexity measure -- some function of the amount of a particular resource used by a program in processing an input. Typical resources would be execution time, core, I/O devices, and channels.

Any resource whose use is independent of previous and future usage can be handled by the theory. This condition includes time but excludes space complexity. For a specific measure, the complexity of the basic programming elements can be determined and used to compute the complexity of an arbitrary program with a particular input. Because this method gives little information about the general complexity behavior of the program, another approach is developed.

This new approach analyzes the complexity of a program with respect to a valid set of inputs -- a finite set of legitimate, halting inputs. A program equation is developed to make the transformations undergone by the inputs more explicit. Using the equation, the input set is partitioned into classes of constant complexity. The classes are used to compute maximum, minimum, and expected complexities of the program on the input set.

Several equivalence relations are defined, relating different programs by their complexity. Complexity is also discussed in terms of concatenation and functional equivalence of programs.

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for the degree of Master of Science, September 1971.

Table of Contents

Chapter I. Introduction

Within the past ten years, there has been an increased interest by computer scientists and mathematicians in the theory of computational complexity. This theory is concerned with measuring the difficulty of computing functions and with studying the properties of measures of computational difficulty. Most of the work done in this field has remained within the domains of recursive function theory and the analysis of Turing machine computation of functions. (See, for example, Hartmanis and Hopcroft [1] for an overview of complexity theory.) One area of complexity theory that has not received much attention is the analysis of functions represented by computer programs. The current research is directed towards this area.

We attempt to apply the basic principles of computational complexity theory to algorithms which are implemented in typical programming languages. One of the basic ideas of this complexity theory of programming languages is to view the complexity behavior of a program over a finite set of "valid inputs", rather than over some infinite domain or for only one input. A "valid input" is one for which the program in question halts and which the program is actually intended to process. Looking at the complexity for all the elements in a set of this type will enable us to get a better picture of the complexity behavior of the program. We will also be able to define some relationships between different programs if the complexities

of their elements relate in certain ways.

1. Functions, Algorithms, Programs

We have mentioned the concept of computational complexity with regard to functions, algorithms, and programs without clearly defining these three terms and explaining the differences between them.

A function defines an association between the objects of one set (the domain of the function) and the objects of another set (the range). The method of determining the object in the range set corresponding to the object in the domain need not be explicitly stated; a set of ordered pairs of the form

(domain element, range element)

completely defines a function. Even if a rule for the function is given (e.g., via a lambda expression [2]) the evaluation of the rule may remain indeterminate. For example, if we have the function

$$\lambda x.\, x+x*x$$

do we perform the addition first or the multiplication?

The computational complexity of a function would have to measure the difficulty of computing the range element given a domain element, no matter what rule was used to determine the range element (more than one rule may specify the same function), or how the rule was evaluated (as long as the evaluation procedure produced the correct answer). Complexity theory of functions is outside of the current discourse.

An algorithm is either the specification of the rule which defines a function and the method for evaluating the rule, or simply the evaluation procedure for a given rule. An algorithm is frequently presented in terms of a flow chart, where each of the nodes of the chart represents a basic operation whose meaning and evaluation are (hopefully!) unambiguous.

The complexity of an algorithm is more basic than that of a function. In the case of an algorithm, we need only examine the specific evaluation procedure defined by the algorithm in order to determine the complexity behavior we wish to observe. However, in our complexity analysis, we will eventually come down to analyzing the basic elements of the algorithm: arithmetic operations, assignments of values to variables, testing conditions, branching, etc. The complexity of these simple operations generally cannot be specified any further. We may choose to express these operations in terms of the corresponding Turing machine operations and deal with Turing complexity. Although this may be adequate in some cases, the complexity of an algorithm on a Turing machine does not give much insight into the complexity of the same algorithm written in a programming language and run on a computer.

A program is the implementation of an algorithm in a particular programming language. If the program is written in the assembly language of a particular machine, we can determine the complexity of the basic elements of the programs in terms of

the characteristics of that machine. If the program is written in a high-level language, the complexity properties of the basic operations would not be completely constrained until we specify which machine the program will run on (and probably how the language translator to be used on the program would work). However, we may choose to leave the complexity in terms of the basic operations of the high-level language.

2. Complexity Measures

The computational complexity of a program may be interpreted in many different ways. A scheme for measuring a specific type of complexity will be called a complexity measure. Basically, a complexity measure is some function of the amount of a particular resource used by a program as it processes a specific input value. This resource might be time, space, CPU usage, channel activity, etc. We might have a program ϕ with input n. Associated with ϕ is a measuring program Φ which is "monitoring" the execution of ϕ. $\Phi(n)$ would tell us the amount of a particular resource used by ϕ to compute $\phi(n)$. Thus the program Φ is measuring the complexity of ϕ.

In the recursive function formulation of complexity theory, a complexity measure is a recursive enumeration of all partial recursive functions ϕ_i, to each of which is associated a step-counting function Φ_i. Φ_i is constrained to satisfy the following two axioms (from Blum [3]):

1. $\phi_i(n)$ is defined iff $\Phi_i(n)$ defined

2. $M(i,n,m) = \begin{cases} 0 \text{ if } \Phi_i(n) \neq m \text{ is a recursive function} \\ 1 \text{ if } \Phi_i(n) = m \end{cases}$

By defined, we mean that a function halts for a particular input.

Uses of Complexity Measures

As we have stated, a complexity measure provides information on the resource usage of a program. As long as programmers have been writing computer programs, they have been concerned with the resource usage of their programs; specifically, they have wanted to know how long their programs would run and how much core they would require. As multiprocessed and time-shared computer systems evolved, programmers wanted to know about the use of system resources other than CPU time and core: channel usage, device usage, secondary storage requirements, supervisor usage, etc. The amount of each of these resources used by a program would constitute a different measure of its complexity.

A theory of computational complexity would give us a method for quantitatively analyzing the complexity behavior of computer programs and for comparing different programs on the basis of this behavior. Hopefully, this theory should be somewhat independent of which type of complexity is being measured, so that the same techniques would be suitable for a number of different resources.

3. Previous Work

There has been little work done in the area of complexity measures for programming languages. Meyer and Ritchie [4]

found some weak bounds on the running time of a class of programs called Loop Programs. These programs compute exactly the primitive recursive functions. However, programs written in most languages will compute recursive functions which are not primitive recursive. Thus, the Loop Program analysis is not general enough.

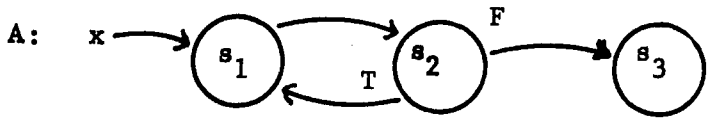
Ramamoorthy [5] studied the time complexity of programs which could be modelled by discrete Markov processes: each decision-making element in the program is statistically independent of all others. He felt that his analysis would be useful for micro-programmed instruction sequences. However, the techniques developed would not work in the case of an arbitrary program.

4. Graph Model of Programs

We will be using the graph model to represent the structure of computer programs. We present here an informal definition of this model. A graph of a program consists of a set of nodes connected by a set of directed arcs. The nodes represent the statements or elements of the program; they will be labelled with a statement identifier or function name. The arcs represent the flow of control in the program. They determine the execution sequence of the nodes. More than one arc may leave a particular node. In this case, each arc will be labelled with a unique selector that determines which node will be executed next.

A successor of a given node is a node which is pointed to by a directed arc from the given node. A predecessor of a given node is a node which points to the given node. A node may precede or succeed itself. A node with no arcs leaving it is a terminal node; it is the last node to be executed. A program graph may have more than one terminal node. The node which is pointed to by an arc that has no node at its other end is the starting node. A graph may have only one such node. The arc leading into the starting node may have an input value or set of values at its other end. The entire graph will usually be labelled with the name of the program.

Thus, the program A consisting of the statements s_1, s_2, s_3 where s_2 is a conditional statement with two possible successors would be represented as:



x is the input value; s_1 is the starting node, s_3 is the terminal node. The successors of s_2 are s_1 and s_3; the predecessor of s_2 is s_1. T and F are selectors for s_2.

Chapter II. Complexity of Arbitrary Programs for Single Inputs

1. Introduction

Before we analyze the complexity behavior of programs over a set of inputs, we will first present methods for determining the complexity of a program with respect to one input. This will involve defining the complexity of the basic programming constructs to be used in the programs, and specifying a set of rules which will enable us to compute the complexity of a group of basic constructs which have been combined. These rules will determine the types of complexity measures for which our methods and techniques will be valid. It happens that these same rules will be sufficient for analyzing complexity behavior over a set of inputs.

2. Constraints on Complexity Measures

We will require that our measures of complexity satisfy three rules. The first two rules are the axioms of Blum presented in Chapter I. The first axiom, $\phi_i(n)$ defined iff $\Phi_i(n)$ defined, implies that the measuring function (program) Φ_i must depend on the entire computation of ϕ_i on input n; if this were not true, then Φ_i might produce an answer even though ϕ_i did not halt. This axiom also implies that when ϕ_i terminates, we have all the necessary information to determine the complexity.

The second axiom,

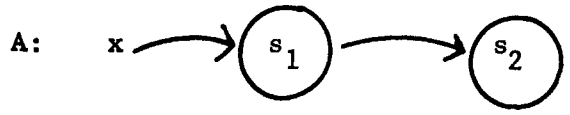
$$M(i,n,m) = \begin{cases} 0 \text{ if } \Phi_i(n) \neq m \\ 1 \text{ if } \Phi_i(n) = m \end{cases} \quad \text{is a recursive function}$$

states that we can always tell if ϕ_i operating on input n will have

complexity m. In the case of time complexity we could let our program ϕ_i with input n run until it had executed for m time units. If ϕ_i halted, M(i,n,m) = 1. If ϕ_i continued to execute, M(i,n,m) = 0.

The third rule is the linearity constraint. It is this rule which allows us to find the complexity of a group of basic elements which are formed into a structured program. The constraint may be stated as follows:

Let A be a program with input x, such that A can be divided into two segments s_1 and s_2 where s_1 is the predecessor of s_2. A graph for A would be:



We can represent the complexity of A with input x as C(A,x). Similarly the complexity of s_1 is $C(s_1,x)$. However, s_2 does not have input x but rather some transformation upon x, induced by s_1. We can represent the input to s_2 as $s_1(x)$. The complexity of s_2 is then $C(s_2,s_1(x))$. Linearity requires that for all x for which A halts,

$$C(A,x) = C(s_1,x) + C(s_2,s_1(x))$$

Another way to state the linearity constraint is that any use of the resource in question must be independent of how much was used previously or how much will be used in the future, but dependent upon transformations of the input.

Time of execution satisfies this constraint. Other resources which also do are the number of calls to the supervisor program, device usage, and channel usage (from the point of view of the number of times the channel is used). One resource that does not generally satisfy the constraint is the amount of core used by a program. Since core may be shared, it may be available for different uses at different times. In the previous example, some of the space needed for s_2's computations may be done in s_1's area. Thus,

$$C(A,x) \leq C(s_1,x) + C(s_2,s_1(x))$$

However, if space is never reused, then space complexity may be incorporated into the general theory. For any resource which obeys this constraint, the methods to be presented may be used to determine the complexity of a program with regard to the use of that resource.

3. Complexity of Basic Program Elements

Our goal is to be able to determine the complexity of an arbitrarily structured program. First, we will define the complexity of the basic elements of our language. This language will not be any specific one but will be representative of modern high-level languages. Below, we list one possible set of basic elements. Naturally, we cannot include every possible program construct; however, those listed are found in many languages.

Arithmetic operations

Assignment

Transfer of control

Conditionals

Iteration

Function calls

I/O and supervisor calls

In discussing each of these elements, we will use time as a sample resource. Of course, "time" may be expressed in microseconds, CPU cycles, or any other units. Other complexity measures may be handled similarly.

Arithmetic Operations

This category includes the common mathematical operations. The time complexity of any of these operations is just the time required to execute it. If we are dealing with an assembly or machine language program, we may express the time in terms of the instruction execution time of the corresponding machine. If we are writing in a high-level language, and if we know the computer which will execute the machine code resulting from our program, we may express the time complexity in a similar manner (ignoring any compiler optimization).

If we do not know which machine will execute our program, we may choose to measure the complexity in terms of the number of additions, the number of multiplications, and the number of other independent operations. We may think of having an n-dimensional "complexity vector", where n is the number of independent operations. The "unit vector" for each of the dimensions is the com-

plexity of a basic operation; the coefficient of this unit vector is the number of such operations which have occurred. We can reduce this vector only if we express the unit complexities in terms of something else, such as the machine instruction times of a specific computer. Then we can obtain one value for the complexity, as we did in the case of an assembly language program.

Note that we are treating the complexity of these operations as having a fixed value, independent of the value of the operands. If this assumption were not true, we would have to examine the sub-operations which form an operation until we found some constructs which were complexity-invariant. We will need this condition when we examine complexity for a set of inputs.

Assignment; Transfer of Control

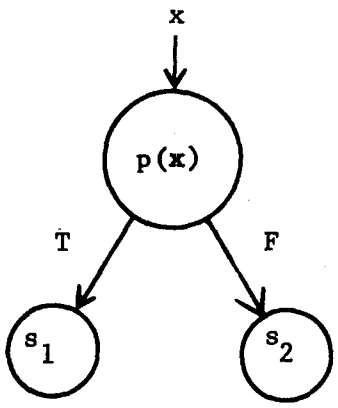
By assignment, we mean the assigning of the value of one variable to another. Transfer of control is the familiar BRANCH or GOTO statement. These two constructs may be treated in the same way as the arithmetic operations: either their complexity may be expressed in terms of instruction execution time or they may be treated as two of the independent operations. We also assume that the complexity of these operations is a fixed value.

Conditionals

The prototype of our conditional statement will be:

IF p(x) THEN s_1 ELSE s_2

p is some test on the value of x which does not change this value. If this test is satisfied (p(x) TRUE), then s_1 will be executed; otherwise s_2 will be executed. We can represent the structure of the conditional by the following graph model:



Using the linearity condition and noting that p does not change the value of x, the complexity of our conditional will be:

$$C(cond,x) = C(p,x) + \begin{cases} C(s_1,x) & \text{if p(x) TRUE} \\ C(s_2,x) & \text{if p(x) FALSE} \end{cases}$$

p, s_1, and s_2 may all represent complex constructions, which can be broken down to basic elements.

Iteration

In most languages, the programmer has the ability to execute a section of program repeatedly, depending upon certain conditions. This is the basis of the iteration statement. We usually have a variable, defined only for the iteration construction, whose value is incremented from a lower limit to some upper

limit while the statements within the bounds of the iteration statement (the body) are executed for each increment of the variable. Some languages allow additional features: multiple ranges for the control variable, negative increments, negative lower limits, attaching a conditional test to the iteration, and others. Examples of the iteration construction are the FORTRAN DO statement, the ALGOL FOR statement, and the PL/I DO statement.

We will use a particularly simple form of iteration statement. We will retain only the concept of executing a body of statements for a certain number of times. This is the LOOP block and has the following form:

```
LOOP N
{body}
END
```

The semantic interpretation of the LOOP block is that the body is executed N (contents of N) times in succession. Changes to N within the body do not affect the number of times that the body is executed. The body may contain other LOOP blocks; thus they may be nested to any level.

The complexity of a LOOP block may be interpreted in several ways. We may view the LOOP structure as equivalent to N physical copies of the body of the block. We may then compute the complexity using linearity. Alternatively, since the LOOP block is usually the feature of a high-level language, we may

examine its translation in machine language. This will involve the initialization of a dummy variable, the body of the block, the incrementing of the variable, a test to see if we have exceeded the number of iterations, and a transfer to the beginning of the body. The complexity of the block could then be computed using the complexity of these elements and the linearity condition.

We will use the simpler interpretation of complexity. If we denote the body of the LOOP block by s, and assume that all statements within s are a function of the variable x, then the LOOP block

```
LOOP N
s
END
```

will have complexity

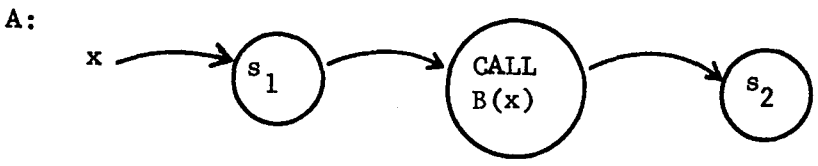
$$C(\text{LOOP-BLOCK},x) = C(s,x) + C(s,s(x)) + C(s,s^2(x)) + C(s,s^3(x)) + \ldots + C(s,s^{N-1}(x))$$

where s^i denotes the functional composition of s with itself i times.

Function Calls

A function or subroutine call involves a call to the subprogram executed in the calling program, and the execution of the subprogram itself. We will assume that the call statement is a basic construct similar to an arithmetic operation and

that its complexity is independent of its argument; furthermore, we will assume that it does not change the value of its argument. The value of the call statement argument, and hence the input to the subprogram, will be some transformation of the original input to the calling program. As an example, consider the program A with input x which contains a call to the function B with argument x (actually some transformation on the original value of x). A could be represented in graphical form as:



The complexity of A would then be

$$C(A,x) = C(s_1,x) + C(CALL) + C(B,s_1(x)) + C(s_2, B \cdot s_1(x))$$

All occurances of x in this expression denote the original input value of x.

I/O and Supervisor Calls

We can view these constructs in the same light as arithmetic operations. For a particular machine, the time complexity of these elements is simply the time needed to execute them. In a high-level language, we may treat them as independent

operations.

Types of Complexity

Certain elements have a constant complexity which does not depend on their inputs. This group includes arithmetic operations, assignment, transfer, and the supervisor operations. Conditional statements have a constant complexity for those inputs for which p(x) is true, and another constant complexity for those inputs for which p(x) is false. This assumes that the elements composing p, s_1, and s_2 are all of constant complexity. Finally, we have LOOP's and subroutine calls which have a complexity dependent upon the input and number of iterations for the LOOP block and the input for the subroutine or function.

If certain supervisor operations have different complexities for different inputs, we may treat them as subroutines.

4. Complexity of Arbitrary Programs

Having defined the complexity of our basic programming constructs, we can use these complexities and the linearity condition to find the complexity C(A,x) of any program A with any input x for which A halts. (We may easily extend our results to include the case of programs with multiple inputs.) This approach, however, does have shortcomings.

The complexity of a program with respect to one input will not usually give us much information about the general complexity behavior of the program. We will know even less if the program does not halt for that input. If we wish to learn more

about the complexity behavior, we will need to repeat our calculations for a large set of inputs. Although the program may exhibit the same complexity behavior for many different inputs, we will not be able to take advantage of this relationship because we are dealing with each input individually. The domain of input values is arbitrarily large until we bind our program to a particular machine or language specification. Because of the nature of this domain, we cannot bound the values of $C(A,x)$ since A defines an arbitrary partial recursive function. Finally, we have no facility for comparing the complexity behavior of two different programs other than inputting the same value to both programs and calculating the resultant complexity.

5. The Set Approach

The remaining chapters will use the work on single input complexity to develop another approach to program complexity. This approach examines the complexity behavior of a program over a finite set of inputs which have certain useful properties. The set approach will make it easy to examine such behavior as the expected value of complexity and maximum and minimum complexity; the complexity structure of a program will become more apparent. Also, the set approach will enable us to examine complexity relations between different programs.

Chapter III. Program Equations and Set Complexity

1. Introduction

Having defined the complexity of the basic program elements and developed methods for determining the complexity of a program with respect to one input, we are ready to deal with program complexity with respect to a set of inputs. We develop the concepts of the program equation and a valid set of inputs to aid in the complexity analysis. We make some simplifying assumptions about the type and number of inputs to our programs and the type of components in these programs. In the next chapter we remove these restrictions to obtain a more general model. An explanation of the mathematical notation used in this and later chapters will be found in the appendix.

2. Input Sets

We will assume a simplified form of input structure for our programs. These programs will have only one input, which will be a non-negative integer; further, we will assume that all operations upon this input result in non-negative integer values. Given a program A which satisfies these conditions, let U_1 be the set of non-negative integer inputs for which A halts, and which are valid inputs to A. By a valid input, we mean an input which A is actually intended to process. Thus, if A is meant to process only even non-negative integers, then U_1 contains only these integers, though A may halt for some odd integers or, in fact, for all odd integers.

Next, let U_2 be the set of allowable non-negative integer values for the machine on which A is running or for the language in which A is written. (If both the language and the machine limit the set of values, we will use those conditions which are more restrictive.) U_2 may be quite large, but it is always finite. We can now define a valid input set X to the program A as:

$$X = U_1 \cap U_2$$

We see that X is finite and for all inputs x in X, A halts.

We may be interested in examining only some of the valid inputs to A at any particular time. If $X' \subseteq X$, we will say that X' is also a valid input set to A. X' has the same properties as X except that it does not contain all the valid halting inputs for A. We will refer to X as the maximal valid input set to program A for a particular U_2 - i.e., for a particular machine or language realization.

3. Program Equations

We now define a method for obtaining an equation representation for a program. We will initially assume that the program contains no LOOP blocks or subroutine calls, and has only one input. These restrictions will be removed later.

The concept of a program equation is based on the work of Zeiger [6]. He derives some relationships between programs, polynomials, and power series. We start the derivation of

the equation by putting our program A into graphical representation. With each node of the graph, we can associate a function on the inputs to that node. Since we have temporarily eliminated LOOP's and subroutines, we have remaining arithmetic operators, assignment, transfer, conditionals, and supervisor calls. Transfer does not change any values. Assignment is a type of arithmetic operation, and supervisor calls change values of variables in specified ways. Thus, we have two general types of functions: arithmetic and conditional.

An arithmetic function f maps a set X into another set F, defined as:

$$F = f(X) = \{ f(x) \mid x \in X \}$$

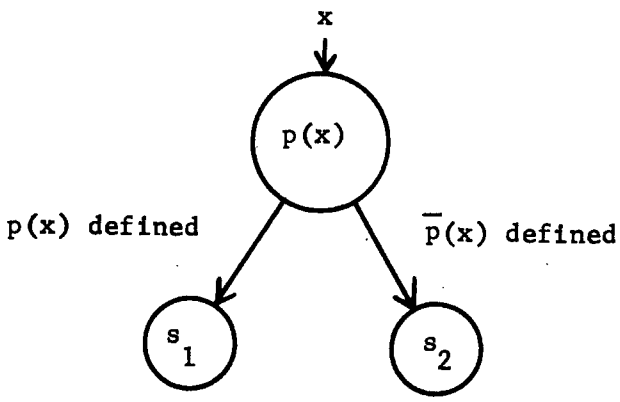
A conditional function p maps X into a set P. We will say that $p(x) = x$ if x satisfies the predicate p; otherwise p(x) is undefined. Therefore, p acts as an identity function for those inputs which satisfy its associated conditional test. We can see that p(x) is defined if and only if $p(x) = x$. Then

$$P = p(X) = \{ x \in X \mid p(x) \text{ is defined} \}$$

We are using the conditional function in a different sense than the conditional statement of the preceding chapter. That statement had the form

IF p(x) THEN s_1 ELSE s_2

p(x) is assumed to take either the value TRUE or FALSE. If p(x) = TRUE, we execute s_1; otherwise, we execute s_2. In the case of a conditional function, we have the following situation:



We define the function $\bar{p}(x)$ as follows:

$$\bar{p}(x) \text{ defined iff } p(x) \text{ undefined}$$

If p(x) is defined, its value is x; we take the arc with the appropriate selector and execute s_1 with input x. We cannot execute s_2 because the selector leading to this node was not satisfied.

Conversely, if p(x) is undefined, $\bar{p}(x)$ is defined; its value is x. We then execute s_2 with input x. One case or the other (but not both) must happen. We also have the relation

$$\bar{\bar{p}}(x) = p(x)$$

so that if our conditional function is $\bar{p}(x)$, our selectors will remain the same. We will abbreviate "p(x) defined" by "T"

(TRUE) and "$\bar{p}(x)$ defined" by "F" (FALSE). This is in keeping with the standard notation of program conditionals.

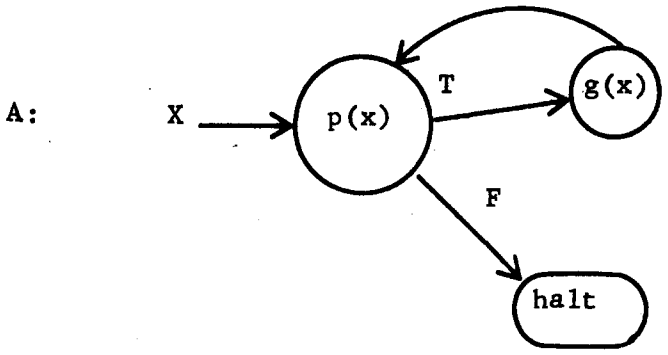
Having specified the function associated with each node of the program graph, we define the equation of the program as the summation of all possible functional paths from the starting node of the graph to any terminal node. A functional path is defined as the composition of the functions associated with the nodes which comprise the path. The resulting composed function is applied to the set of inputs. The sum of these functional applications is set equal to the function defined by the entire program applied to the set of inputs.

Given a program A with input set X, its equation would be

$$A(X) = \sum_{i=0}^{\infty} f_i(X)$$

where each f_i is a composition of simpler functions. The summation is of infinite extent because an arbitrary program graph contains an infinite number of paths.

We illustrate these concepts with a sample program:



The simplest path through A starts at p and takes the F arc to the halting node. We will assume that this latter node has no functional or complexity significance. If we recall that F is equivalent to "$\overline{p}(x)$ defined", the first term of A(X) will be

$$f_0(X) = \overline{p}(X) = \{ x \epsilon X \mid \overline{p}(x) \text{ defined} \}$$

The next simplest path starts at p, takes the T arc to g, returns to p, and then takes the F arc to the halting node. The composition of the functions encountered in this path is (with the right-most function encountered first)

$$f_1 = \overline{p} \cdot g \cdot p$$

This function applied to X yields the next term of A(X):

$$f_1(X) = \{ g(x) \mid x \epsilon X \ \& \ p(x) \text{ defined} \ \& \ \overline{p}(g(x)) \text{ defined} \}$$

Continuing in this manner we get the complete equation:

$$A(X) = \overline{p}(X) + \overline{p} \cdot g \cdot p(X) + \overline{p} \cdot (g \cdot p)^2(X) + \ldots$$
$$= \sum_{i=0}^{\infty} \overline{p} \cdot (g \cdot p)^i(X)$$

where $(g \cdot p)^0$ = the identity function. There is no unique correspondence between a particular index and a particular functional path. In this case, we have calculated functions in order of

increasing path length. It happened that

$$f_i = \bar{p} \cdot (g \cdot p)^i$$

However, any other one-to-one correspondence between the f_i and the functional paths would work. We will conclude this example later in the chapter.

The application of f_i to X results in a set of values. We define

$$F_i = f_i(X)$$

The +'s in our equation are to be interpreted as set union. Then

$$F_i \subseteq A(X) \text{ and } \bigcup_{i=0}^{\infty} F_i = A(X)$$

Each f_i is generally composed of arithmetic and conditional functions. Suppose $f_i = g_n \cdot \ldots \cdot g_1$. We will say that $f_i(x)$ is defined if and only if for all conditional functions $g_j \in \{ g_n, g_{n-1}, \ldots, g_1 \}$, g_j applied to its argument is defined; i.e., $g_j(g_{j-1} \cdot \ldots \cdot g_1(x))$ is defined. Then $f_i(x) \in F_i$ if and only if $f_i(x)$ is defined. Now we can define the set of elements in X which exactly yield each F_i:

$$X_i = \{ x \in X \mid f_i(x) \text{ is defined} \}$$

We then have the relation $F_i = f_i(X_i)$.

Simplification of the Equation

We can reduce our equation if we require that X be a valid input set to A. We then have that X is finite and for all inputs in X, A halts. Since X is finite, there are only a finite number of non-empty X_i. Let m be the greatest integer for which $X_m \neq \phi$ (the empty set); then for all $i>m$, $X_i=\phi$ and $F_i=\phi$. We then have:

1. $A(X) = \sum_{i=0}^{m} f_i(X)$
2. $\bigcup_{i=0}^{m} X_i = X$ (it may be that some of the X_i are empty; this will occur if the corresponding F_i is empty)
3. $X_i \cap X_j = \phi$ if $i \neq j$ (because programs are deterministic, any input undergoes only one transformation)
4. Each f_i is composed of only a finite number of subfunctions since A halts for all x in X.

Thus, given a valid input set, we can reduce our program equation to a finite number of terms, each term of finite extent. We also get a partition $\{ X_0, X_1, \ldots, X_m \}$ of our original input set.

4. Complexity Equivalence Classes

The subsets of X, $\{ X_i \}$, have the following property: for all $x \in X_i$, the functional path associated with x is the same. Since we have allowed no LOOP blocks or subroutines in our programs, and since all x in X_i take the same branch from every

conditional test, the complexity of this functional path is constant. This complexity is completely defined in terms of the functions in the path and has a value $C(A,x)$, $x \epsilon X_i$. X_i will be called a complexity equivalence class. Associated with each X_i will be an equivalence class complexity - $C(A,X_i)$. By definition,

$$C(A,X_i) = C(A,x),\ x \epsilon X_i$$

The complexity information for program A with input set X is contained in the two sets

$$\{ X_0, X_1, \ldots, X_m \}$$

$$\{ C(A,X_0),\ C(A,X_1), \ldots, C(A,X_m) \}$$

We observe that many programs do not treat every input differently. Often, many inputs will be similarly processed. Therefore, we are led to believe that the number of equivalence class will often be smaller (though not always much smaller) than the number of distinct input values. In this case, we have a (relatively) compact description of the complexity behavior of A.

Uses of Complexity Equivalence Classes

The complexity equivalence classes may be used to calculate the following quantities with respect to a particular input set:

1. Expected value of complexity

2. Maximum value of complexity

3. Minimum value of complexity

In a later chapter we will show how these classes are useful in determining relations between different programs.

Expected Complexity Value

Expected complexity value will tell us the average resource usage by a program over a particular input set. The expected value of a function g over a discrete-valued domain X is (see, for example, Drake [7]) the sum of the products of values of the function at points in the domain and the probability that the particular point will be chosen:

$$E(g,X) = \sum_{x \epsilon X} g(x) \cdot Pr(x)$$

If C(A,X) is the expected value of the complexity of A with input set X,

$$C(A,X) = \sum_{x \epsilon X} C(A,x) \cdot Pr(x)$$

Since $\bigcup_{i=0}^{m} X_i = X$,

$$C(A,X) = \sum_{X_i} \sum_{x \epsilon X_i} C(A,x) \cdot Pr(x)$$

But for all $x \epsilon X_i$, $C(A,x) = C(A,X_i)$; and since $x_1 \epsilon X_i$ and $x_2 \epsilon X_i$ are independent events,

$$Pr(X_i) = \sum_{x \epsilon X_i} Pr(x)$$

Then,

$$\sum_{x \epsilon X_i} C(A,x) \cdot Pr(x) = \sum_{x \epsilon X_i} C(A,X_i) \cdot Pr(x) =$$

$$C(A,X_i)(\sum_{x \epsilon X_i} Pr(x)) = C(A,X_i) \cdot Pr(X_i)$$

$$\therefore \quad C(A,X) = \sum_{X_i} C(A,X_i) \cdot Pr(X_i)$$

The probabilities $Pr(X_i)$ are based on any method used in selecting the inputs to A, and take into account any *a priori* knowledge of the selection method. If the selection of inputs is random,

$$Pr(X_i) = | X_i | / | X |$$

If $X_i = \phi$, then for any selection method, $Pr(\phi) = 0$. The value of C(A,X) is not changed by averaging over empty sets.

Maximum Value of Complexity

The maximum complexity is an upper limit on the usage of a particular resource for any execution of a program, given an input value from the input set in question. To compute this quantity, we need only find the maximum value of $C(A,X_i)$ over the X_i. We must remember that some X_i may be empty; so we only look at $C(A,X_i)$ for non-empty X_i. The maximum complexity will be denoted $C_{max}(A,X)$.

Minimum Value of Complexity

The minimum complexity is the corresponding lower limit of resource usage by the program. It is the minimum of $C(A,X_i)$ over the non-empty X_i. It will be denoted $C_{min}(A,X)$.

$C_{max}(A,X)$ and $C_{min}(A,X)$ are important quantities. They say that any execution of the program will require $C_{min}(A,X)$ of resources, but never more than $C_{max}(A,X)$, when inputs are taken from X.

5. Conclusion of Example Program

Returning to our sample program introduced earlier in this chapter, we will specify the input set and the functions comprising the program:

$X = \{ 1,2,3,\ldots,10 \}$

$g(x)$: $x:=2x$ (x is assigned the value 2x)

$p(x) = x$ iff $x \leq 5$

$\bar{p}(x) = x$ iff $x > 5$

$A(X) = \sum_{i=0}^{\infty} f_i(X)$

$f_0 = \bar{p}$ $\quad f_1 = \bar{p} \cdot (g \cdot p)$ $\quad f_2 = \bar{p} \cdot (g \cdot p)^2 \ldots$

$F_0 = f_0(X) = \{ x \in X \mid \bar{p}(x) \text{ is defined} \} = \{ 6,7,8,9,10 \}$

$X_0 = \{ 6,7,8,9,10 \}$

$F_1 = f_1(X) = \bar{p}(\{ 2,4,6,8,10 \}) = \{ 6,8,10 \}$

$X_1 = \{ 3,4,5 \}$

$F_2 = f_2(X) = \bar{p}(\{ 4,8 \}) = \{ 8 \}$

$X_2 = \{ 2 \}$

$F_3 = f_3(X) = \bar{p}(\{ 8 \}) = \{ 8 \}$

$X_3 = \{ 1 \}$

$\therefore \bigcup_{i=0}^{3} X_i = X$

$\therefore \ (\forall n \geq 4)\ [F_n = X_n = \phi]$

$\therefore \ A(X) = \sum_{i=0}^{3} f_i(X)$

$C(A,X_0) = C(A,x), \ x \epsilon X_0$

Note that the complexity of the conditional test (p itself) is the same whether p(x) or $\bar{p}(x)$ is defined. Therefore,

$C(\bar{p},x) = C(p,x) = c_p, \quad \forall x \epsilon X$

$\therefore C(A,X_0) = c_p$

$C(A,X_1) = C(A,x), \quad x \epsilon X_1$

$C(A,x) = C(p,x) + C(g,x) + C(\bar{p},g(x))$

$= c_p + c_g + c_p = 2c_p + c_g$

$C(A,X_2) = 3c_p + 2c_g$

$C(A,X_3) = 4c_p + 3c_g$

To determine C(A,X), the expected complexity, we need to know the values of $Pr(X_i)$. Let us assume a random distribution. Then

$$Pr(X_i) = | X_i | / | X |$$

$$C(A,X) = \sum_{i=0}^{3} C(A,X_i) \cdot Pr(X_i) =$$

$$\frac{5}{10} c_p + \frac{3}{10} (2c_p + c_g) + \frac{1}{10} (3c_p + 2c_g) + \frac{1}{10} (4c_p + 3c_g)$$

$$= \frac{9}{5} c_p + \frac{4}{5} c_g$$

$$C_{max}(A,X) = C(A,X_3) = 4c_p + 3c_g$$

$$C_{min}(A,X) = C(A,X_0) = c_p$$

6. Summary

We have developed several important concepts for the complexity study of programs - valid input sets, an equation representation of a program, complexity equivalence classes, and equivalence class complexities. We have seen the use of these concepts in the analysis of complexity behavior. Finally, we have shown how the equivalence classes and their associated complexities have given a compact and orderly representation of the complexity information for a program over a set of inputs.

Chapter IV. Complexity of Advanced Constructs and Input Schemes

1. Introduction

In this chapter we extend our complexity analysis procedures to include programs with additional programming features, such as subroutine calls and LOOP blocks, and also programs with more than one input. We also discuss input data types other than the non-negative integers and finally variable length inputs.

2. Subroutine and Function Calls

Let us assume that we have a program A with valid input set X. A has the finite equation representation

$$A(X) = \sum_{i=0}^{n} f_i(X)$$

We can express the j^{th} term, f_j, as a composition of subfunctions:

$$f_j = g_m \cdot g_{m-1} \cdot \ldots \cdot g_1$$

Suppose g_k is a call to the program B. Let Y be the maximal valid input set to B. The equation for B is then

$$B(Y) = \sum_{i=0}^{p} h_i(Y)$$

B will be called from A at point g_k within f_j with a set of values

$$X' = g_k \cdot \ldots \cdot g_1(X)$$

where the series of composed functions is the transformation applied to the elements of X. If we assume that A is working correctly, then all values passed from A will be valid inputs to B. Thus $X' \subseteq Y$. The equation for B(X') will have, at most, as many terms as that for B(Y). Then

$$B(X') = \sum_{i=0}^{q} h_i(X'), \quad q \leq p$$

To obtain the complexity equivalence classes for A with input set X, we cannot simply calculate the X_i corresponding to the $F_i = f_i(X)$. The reason is that there is no unique complexity associated with X_j since f_j contains a call to B which has q+1 distinct functional paths associated with it. To get the true equivalence classes for A, we will have to substitute the terms of B(X') into A(X) between g_{k+1} and g_k in f_j. Assuming no other function calls in A, this procedure will yield an expanded set of functions f_i', each having a unique functional path. Any value computed in B and returned to A will be available to g_{k+1}. This expansion yields the following equation for A:

$$A(X) = \sum_{i=0}^{j-1} f_i(X) + g_m \cdot \ldots g_{k+1} \cdot \left(\sum_{i=0}^{q} h_i \right) \cdot g_k \cdot \ldots \cdot g_1(X) + \sum_{i=j+1}^{n} f_i(X) = \sum_{i=0}^{n+q+1} f_i'(X)$$

We can now calculate the complexity equivalence classes

$$X_i = \{ x \in X \mid f_i'(x) \text{ is defined } \}$$

and the associated class complexities $C(A,X_i)$.

We are assured of finding a new finite representation for A(X) because:

1. A(X) (unexpanded) has a finite number of terms.
2. B(Y) has a finite number of terms.
3. A is assumed to be operating correctly so that $X' \subseteq Y$. Therefore, B(X') has a finite representation. Introducing B(X') into A(X) yields a new finite representation for A(X).

This method is general enough to handle multiple levels of subroutine calls, multiple calls to the same subroutine from the same calling program, and recursive subroutine calls. To illustrate multiple levels of calls, suppose that in the previous example, B called a program C with input set Y'. We would find the equation for C(Y'), substitute this into B(X'), and substitute the expanded B(X') into A(X). We know that the process of subroutine calls must terminate since A halts for all input values in X.

For the case of multiple calls from the same program, assume A called B twice, once with input set X' and once with set X''. We would use our analysis procedure to find B(X') and B(X'') and substitute these into A(X).

Suppose A calls itself as a subroutine. These recursive calls must terminate because of the halting condition. If the set of values passed from A to itself is X', we know that $X' \subseteq X$. After finding A(X'), we substitute it into A(X).

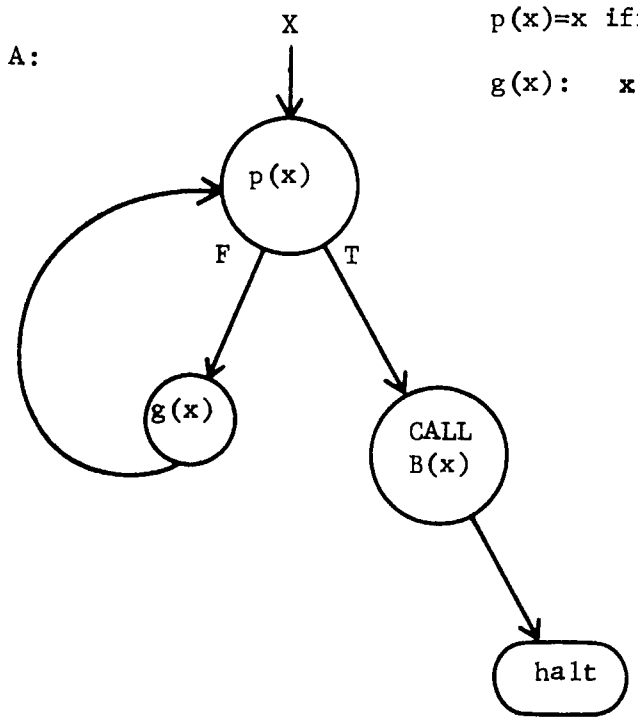
Example of Subroutine Call

We will consider a program A with input set X which calls a program B with maximal input set Y.

$X = \{ 1,2,3, \ldots,9 \}$

$p(x)=x$ iff $x \equiv 0 \bmod 3$

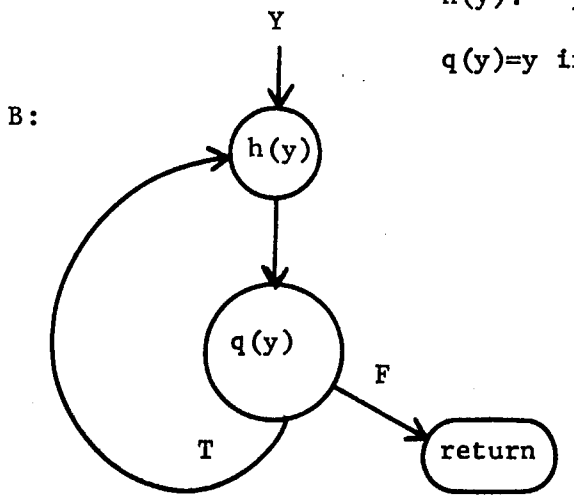
g(x): $x := x+1$



$Y = \{ 3,6,9,12,15, \ldots,3 \cdot n \}$

h(y): y:=y/3

q(y)=y iff $y \equiv 0 \bmod 3$



Denoting the node "CALL B(x)" by "b", we have the following equations:

$$A(X) = b \cdot p(X) + b \cdot p \cdot g \cdot \bar{p}(X) + b \cdot p \cdot (g \cdot \bar{p})^2(X) + \ldots$$

$$= \sum_{i=0}^{\infty} f_i(X)$$

$$B(Y) = \bar{q} \cdot h(Y) + \bar{q} \cdot h \cdot q \cdot h(Y) + \bar{q} \cdot h \cdot (q \cdot h)^2(Y) + \ldots$$

$$= \sum_{i=0}^{\infty} h_i(Y)$$

Using our analysis on just the equation for A(X), we get the following classes:

$$X_0 = \{ 3,6,9 \} \qquad X_1 = \{ 2,5,8 \} \qquad X_2 = \{ 1,4,7 \}$$

$$A(X) = \sum_{i=0}^{2} f_i(X)$$

However, since the inputs in any X_i undergo different transformations in B, these are not complexity equivalence classes. We will have to substitute B(X') into A(X) wherever B is called from A.

Let X'_0 be the set of inputs to B when B is called from f_0. Then

$$X'_0 = b \cdot p(X) = \{ 3,6,9 \} \subseteq Y$$

$$H_0 = h_0(X'_0) = \{ 1,2 \} \qquad X'_{01} = \{ 3,6 \}$$

$$H_1 = h_1(X'_0) = \{ 1 \} \qquad X'_{02} = \{ 9 \}$$

Since $X'_0 = X'_{01} \cup X'_{02}$,

$$B(X'_0) = \sum_{i=0}^{1} h_i(X'_0) = \bar{q} \cdot h(X'_0) + \bar{q} \cdot h \cdot q \cdot h(X'_0)$$

Similarly, let X'_1 be the set of inputs to B when it is called from within the function f_1. Then,

$$X'_1 = f_1(X) = \{ 3,6,9 \} = X'_0$$

$$B(X'_1) = \sum_{i=0}^{1} h_i(X'_1)$$

Finally we let X'_2 be the inputs to B when it is called from f_2.

$$X'_2 = f_2(X) = \{ 3,6,9 \} = X'_0$$

$$B(X'_2) = \sum_{i=0}^{1} h_i(X'_2)$$

We can now substitute the three equations for B into the appropriate places in A(X) to get the expanded version of this equation:

$$A(X) = B(X_0')\cdot f_0(X) + B(X_1')\cdot f_1(X) + B(X_2')\cdot f_2(X)$$

$$= h_0\cdot f_0(X) + h_1\cdot f_0(X) + h_0\cdot f_1(X) + h_1\cdot f_1(X)$$

$$+ h_0\cdot f_2(X) + h_1\cdot f_2(X)$$

$$= \sum_{i=0}^{5} f_i'(X)$$

Each of the classes X_i, i=0,1,...5, now corresponds to a set of inputs with a single complexity value; each X_i is a complexity equivalence class.

3. LOOP Blocks

We have discussed the LOOP block in a previous chapter and defined the complexity of this construction as follows:

If the body of the block is denoted by s and all statements within s are a function of x, then the LOOP block

```
LOOP N
s
END
```

will have complexity

$$C(s,x) + C(s,s(x)) + C(s,s^2(x)) + \ldots + C(s,s^{N-1}(x))$$

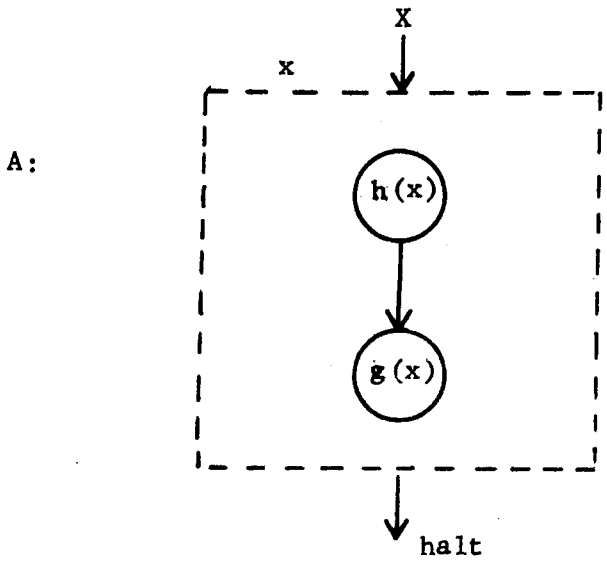
We can now include LOOP's into our theory of program equations and complexity equivalence classes. We will illustrate the

handling of loops by several examples.

Suppose we have the following program A with input x:

```
LOOP x
h(x)
g(x)
END
```

If A has a valid input set X, we may represent A in graph format as follows:



The dotted lines delineate the scope of the LOOP block. The "x" just outside the box defines x as the iteration control variable.

The equation for A(X) will have to indicate that g and h will be executed a different number of times for each value of

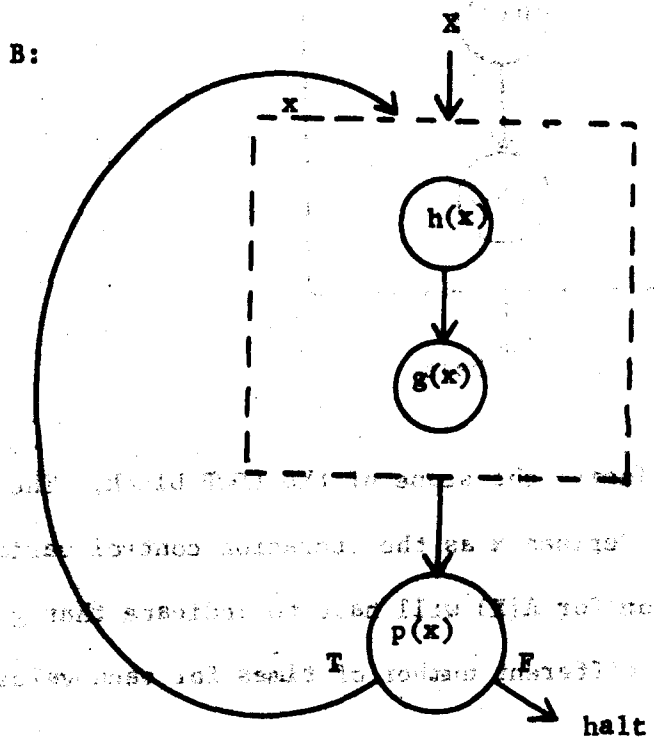
x in X. Assuming no attributable complexity to "END", the equation is

$$A(X) = \sum_{x \in X} (g \cdot h)^{x}(X)$$

If $|X|=n$, this sum will expand to n terms. If g and h have a constant associated complexity, there will be n complexity equivalence classes for A with set X. In this case,

$$X_i = \{ x_i \} \text{ and } C(A,X_i) = C(A,x_i)$$

Let us consider the more complicated program B:



Now the LOOP block may be executed many times, the number of iterations each time (i.e., x) dependent upon previous executions.

To calculate B(X) we use our basic method: find those inputs for which B halts on the first pass through p, on the second pass, etc. Sum these terms to get B(X).

If f_0 is the functional path which terminates on the first path through p,

$$f_0(X) = \bar{p} \cdot \sum_{x \in X} (g \cdot h)^x(X)$$

Similarly, if f_1 denotes the functional path which ends on the second pass through p,

$$f_1(X) = \bar{p} \cdot \sum_{x' \in X'} (g \cdot h)^{x'} \underbrace{\cdot p \cdot \sum_{x \in X} (g \cdot h)^x(X)}_{X'}$$

Continuing in this manner,

$$B(X) = \sum_{i=0}^{\infty} f_i(X)$$

Since X is finite, there exists an n such that for all $i > n$, $f_i(X) = \phi$.

$$\therefore \quad B(X) = \sum_{i=0}^{n} f_i(X)$$

While this is a valid equation for B(X), each $X_i = \{ x \epsilon X \mid f_i(x) \text{ is defined} \}$ does not correspond to a single value of complexity. Each f_i will have to be subdivided by expanding out the summations. The subterms, f_i', will correspond to single values of complexity; and the X_i,

$$X_i = \{ x \epsilon X \mid f_i'(x) \text{ is defined} \}$$

will be complexity equivalence classes.

4. Multiple Inputs

We now extend our complexity analysis procedures to programs with more than one input. We will deal specifically with the case of two inputs. The generalization to programs with n inputs is immediate.

Let A be a program with two inputs, x and y. Analagous to the one input case, we define the set U_1 as

$$U_1 = \{ (x,y) \mid \text{A halts for non-neg. integer inputs x and y; x and y valid} \}$$

We then define U_2 as the set of all ordered pairs of non-negative integer values for the language in which A is written or the machine on which A is running:

$$U_2 = \{ (x,y) \mid \text{x and y are allowable non-neg. integer values} \}$$

Then the maximal valid input set to program A is

$$Z = U_1 \cap U_2$$

As before $Z' \subseteq Z$ is also a valid input set.

We can then proceed to find the functional paths of the program graph from the starting node to a terminal node. The function f_i associated with each path maps ordered pairs (x,y) to (x',y'). f_i is composed of arithmetic and conditional functions. For any $(x,y) \epsilon Z$, we can calculate a complexity C(A,(x,y)) defined by a path through A.

Using the functional paths through A and noting that Z is a valid input set we can obtain the finite equation for A(Z):

$$A(Z) = \sum_{i=0}^{n} f_i(Z)$$

The f_i immediately lead to the complexity equivalence classes

$$Z_i = \{ (x,y) \epsilon Z \mid f((x,y)) \text{ is defined } \}$$

and the equivalence class complexities

$$C(A,Z_i) = C(A,(x,y)),\ (x,y) \epsilon Z_i$$

Before finding the Z_i, we may have had to expand the f_i to take care of any function calls or LOOP blocks. We can now calculate C(A,Z), $C_{max}(A,Z)$, and $C_{min}(A,Z)$ as before.

If we define the sets X and Y as

$$X = \{x \mid \exists y \text{ such that } (x,y) \in Z\}$$

$$Y = \{y \mid \exists x \text{ such that } (x,y) \in Z\}$$

we can easily see that $Z \subseteq X \times Y$. For $(x,y) \in Z$ implies that $x \in X$ and $y \in Y$ which imply that $(x,y) \in X \times Y$. Note that Z is not necessarily equal to $X \times Y$. If, for inputs $x_0 \in X$, $y_0 \in Y$, A does not halt, then $(x_0, y_0) \notin Z$.

Inputs vs. Variables

Although A may have n inputs, it may have m variables (including input variables) internal to it, $m \geq n$. One way to handle the m-n variables which are not inputs is to express them in terms of transformations on the inputs. Thus if y is not an input variable, then before it is used, it must have a value assigned to it. This value is a function of the n-tuple of input values:

$$y = f((x_1, \ldots, x_n))$$

Another way to solve the problem is to view A as having m inputs. Those variables which are not actual inputs have a default value assigned to them, say zero. If A is working correctly, these variables will be assigned new values before they are used for their original default values.

5. Different Data Types

So far, we have limited our discussion to programs which operate on non-negative integers from 0 to intmax - an upper bound imposed by either the language in which A is written or the machine on which A is running. It is not conceptually difficult to extend the complexity analysis procedure to cover other types of input.

In most implementations, negative integers range from -1 to some lower limit: negmax. If we wish to analyze a program A with an integer input (positive or negative), then our maximal valid input set X would still be defined as $U_1 \cap U_2$, but now

$$U_1 = \{x \mid \text{A}(x) \text{ halts and } x \text{ a valid integer input}\}$$

$$U_2 = \{x \mid x \text{ an integer and } \underline{\text{negmax}} \leq x \leq \underline{\text{intmax}}\}$$

We would then procede as before to find the program equation A(X), the complexity equivalence classes X_i, and the subset complexities $C(A,X_i)$.

We may also easily handle character input (the character set on a machine is always finite) and floating point numbers (always finite in number because of the limitations on the magnitude of the exponent and the precision of the fraction). In the most general case, we would have a program with several inputs of different data types. The work on multiple inputs would apply here, generalizing to n-tuples with components of different data types.

6. String Input

We have not yet considered programs which have string or variable length input sequences. An example of such a program would be a compiler whose input is a series of characters in the source language which it is required to translate. There is often only one input variable in such programs; each time a new character is read, the value of that character is placed in the input variable. However, unlike previous programs where current values of variables could be defined in terms of transformations upon older values, the new value of the input variable in the case of string input is not generally definable in terms of previous values. Although we might like to treat this new value as a separate input, the number of such new inputs is indeterminate, so we cannot directly apply our previous methods.

One way out of this problem is to observe that compilers and similar programs usually read their string of inputs until the string is exhausted. Therefore, given an input string of n characters, we know that the program processing this string will read in n distinct input values. If we view an n character string as being n inputs to the program, then each length string defines a different input situation. Thus, when analyzing the complexity of a program with string input (where the string is always read in its entirety), a set of valid inputs would be defined as a subset of all strings of a fixed length, say n. We would then treat the program as having n inputs.

Chapter V. Results in the Complexity Theory of Programming Languages

1. Introduction

We are now ready to investigate some of the properties of the complexity-theoretic concepts previously introduced. We define some complexity relations between programs, define the complexity of concatenated programs, and study equivalence of programs in the light of complexity theory. We prove some results about these areas.

2. Preliminary Definitions

We first define a special form of the complexity equivalence classes of a program; this form will be used in most of the definitions which follow. It is a standard which will be used in comparing equivalence classes of different programs.

Definition: Given a program A with input set X, we define the set $\overline{X}_A$ as follows:

$$\overline{X}_A = \{ X_i \mid X_i \text{ is a complexity equivalence class of A with input set X} \}$$

Definition: $\overline{X}_A$ is said to be in normal form iff

$$(\forall X_i \in \overline{X}_A)$$

$$[X_i \neq \phi \text{ (empty set) and } i \neq j \Rightarrow C(A,X_i) \neq C(A,X_j)]$$

Normal form implies that all the complexity equivalence classes X_i contain at least one element of X and the complexity of any equivalence class, $C(A,X_i)$, is unique. It is this normal form which allows us to compare the set of all equivalence classes of two different programs.

If $\bar{X}_A$ is not in normal form, it may be put in this form by deleting any $X_i = \phi$, and if $C(A,X_i) = C(A,X_j)$, then replacing X_i and X_j by a new equivalence class $X_{ij} = X_i \cup X_j$ where $C(A,X_{ij}) = C(A,X_i)$. Note that the deletion of $X_i = \phi$ or the creation of X_{ij} does not change $C_{max}(A,X)$, or $C_{min}(A,X)$, or $C(A,X)$. In the case of $C(A,X)$, we have

$$X_i = \phi \Rightarrow Pr(X_i) = 0$$

Also if we form $X_{ij} = X_i \cup X_j$, then since $X_i \cap X_j = \phi$,

$$Pr(X_{ij}) = Pr(X_i) + Pr(X_j)$$

$$\therefore C(A,X_i) \cdot Pr(X_i) + C(A,X_j) \cdot Pr(X_j)$$

$$= C(A,X_{ij}) \cdot (Pr(X_i) + Pr(X_j))$$

$$= C(A,X_{ij}) \cdot Pr(X_{ij})$$

If we replace X_i and X_j by X_{ij}, $C(A,X)$ remains the same.

Next we define a shorthand notation for the complexity equivalence of single inputs and equivalence classes of inputs.

Definition: Given a program A, the relation $=_A$ is defined as follows:

For x_1, x_2 valid inputs to A, $x_1 =_A x_2$ iff $C(A,x_1) = C(A,x_2)$

For X_i, X_j equivalence classes of A, $X_i =_A X_j$ iff $C(A,X_i) = C(A,X_j)$

It is easy to see that $=_A$ is an equivalence relation (reflexive, symmetric, transitive) since it is defined in terms of another equivalence relation (=).

Lemma 1: If $\overline{X}_A$ is in normal form and $x_1, x_2 \in X$, then

$x_1 =_A x_2$ iff ($\exists$ unique $X_i \in X_A$ such that $x_1, x_2 \in X_i$)

Proof: ($\Leftarrow$) $x_1, x_2 \in X_i \Rightarrow C(A,x_1) = C(A,x_2) \Rightarrow x_1 =_A x_2$

($\Rightarrow$) $x_1 =_A x_2 \Rightarrow C(A,x_1) = C(A,x_2)$. Suppose $x_1 \in X_j$ and $x_2 \in X_k$. Then $X_j =_A X_k$. By normality of $\overline{X}_A$, $X_j = X_k$. Then $x_1, x_2 \in X_j$, $X_j \in \overline{X}_A$. Since the X_i are disjoint, X_j is unique.

QED

The lemma gives us another statement of the normal form condition - all inputs with the same associated complexity are in the same equivalence class. This result will be useful in later sections.

3. Relations Between Programs with Identical Input Sets

We now turn to relations between programs which have the same valid input set. We will develop a series of equivalence relations between such programs, based on different complexity properties of the programs.

Similarity

The first equivalence relation, which we now define, is also the weakest. It specifies a relation between programs which divide the same input set into the same complexity equivalence classes.

Definition: Given programs A and B, both with input set X and with $\overline{X}_A$ and $\overline{X}_B$ in normal form, A and B are similar on X iff $\overline{X}_A = \overline{X}_B$.

Similarity between two programs on a given input set is equivalent to saying that two inputs to one program have the same complexity if and only if they have the complexity when input to the other program. This may be formally stated in the following theorem:

Theorem 1: A and B are similar on X iff

$$(\forall x_1, x_2 \in X)[x_1 =_A x_2 \Leftrightarrow x_1 =_B x_2]$$

Proof: (=>) Suppose $x_1 =_A x_2$. Then by lemma 1, $x_1, x_2 \in X_i$, $X_i \in \overline{X}_A$ where $\overline{X}_A$ is in normal form. But by similarity, $\overline{X}_A = \overline{X}_B$; then $X_i \in \overline{X}_B$. Therefore, $x_1 =_B x_2$. Likewise, $x_1 =_B x_2 \Rightarrow x_1 =_A x_2$.

($\Leftarrow$) Construct $X_i \in \overline{X}_A$ from all $x \in X$ such that $x =_A x_i$; similarly construct $X_i' \in \overline{X}_B$ from all $x \in X$ such that $x =_B x_i$. Since $x =_A x_i$ iff $x =_B x_i$, $X_i = X_i'$. Continuing for all $x_i \in X$, we see that $X_i \in \overline{X}_A$ iff $X_i \in \overline{X}_B$. Thus $\overline{X}_A = \overline{X}_B$. None of the X_i are empty (because $=_A$ and $=_B$ are reflexive); if $C(A,X_i) = C(A,X_j)$, then by hypothesis $C(B,X_i) = C(B,X_j)$. Therefore in normalized form, $\overline{X}_A = \overline{X}_B$. Thus A and B are similar on X. QED

Although the complexities for the two inputs (x_1 and x_2) are the same in <u>each</u> program, the magnitude of this complexity for the two programs is, in general, different. Thus $x_1 =_A x_2$ and $x_1 =_B x_2$ do not imply that $C(A,x_1) = C(B,x_1)$.

Similarity between two programs is preserved by taking subsets of the original input set, and by intersection of different input sets which induce similarity.

<u>Theorem 2</u>: If A and B are similar on X and also on Y, they are similar on $Z \subseteq X$ and on $X \cap Y$.

Proof: (Subset) Let $|\overline{X}_A| = n$. Define $Z_i = Z \cap X_i$, $\forall X_i \in \overline{X}_A$. Then

$$\bigcup_{i=1}^{n} Z_i = \bigcup_{i=1}^{n} (Z \cap X_i) = Z \cap (\bigcup_{i=1}^{n} X_i) = Z \cap X = Z$$

Also, for $i \neq j$,

$$Z_i \cap Z_j = (Z \cap X_i) \cap (Z \cap X_j) =$$

$$Z \cap (X_i \cap X_j) = Z \cap \phi = \phi$$

Eliminating any $Z_i = \phi$, $\overline{Z}_A = \{Z_i\}$ is in normal form. Since $\overline{X}_A = \overline{X}_B$, $\overline{Z}_A = \overline{Z}_B$. Therefore A and B are similar on Z.

(Intersection) $X \cap Y \subseteq X$. Using the first part of the proof, A and B are similar on $X \cap Y$. QED

Absolute Similarity

Certain pairs of programs may be similar on all valid input sets. We have the following definition:

Definition: A and B are absolutely similar iff A and B are similar on all valid input sets X.

Lemma 2: If A and B are similar on the maximal valid input set X, they are absolutely similar.

Proof: All valid input sets are subsets of the maximal valid input set. By theorem 2, since A and B are similar on the maximal input set, they are similar on all valid input sets. Thus A and B are absolutely similar. QED

Lemma 3: If A and B are absolutely similar, and if A and B are similar on X and on Y, they are similar on $X \cup Y$.

Proof: The union of two valid input sets is also a valid input set. By hypothesis, A and B are similar on all valid input sets. QED

Homomorphism

We define the relational operators $<_A$ and $>_A$ for single inputs and input classes.

Definition: For x_1, x_2 valid inputs to A, $x_1 >_A x_2$ iff $C(A,x_1) > C(A,x_2)$ and $x_1 <_A x_2$ iff $C(A,x_1) < C(A,x_2)$.

For X_i, X_j equivalence classes of A, $X_i >_A X_j$ iff $C(A,X_i) > C(A,X_j)$ and $X_i <_A X_j$ iff $C(A,X_i) < C(A,X_j)$.

Similarity tells us that for B similar to A, $x_1 =_A x_2$ iff $x_1 =_B x_2$. It also implies that $x_1 \neq_A x_2$ iff $x_1 \neq_B x_2$. We cannot say that $x_1 <_A x_2$ implies $x_1 <_B x_2$, nor can we establish any other ordering relation on the complexities of inputs. To do this, we need to define another relation on programs with the same input set which will tell us something more about the relative magnitudes of the equivalence class complexities. This leads to the following definition:

Definition: Suppose A and B are similar on X and we order the equivalence class complexities $C(A,X_i)$ and $C(B,X_j)$ in strictly increasing order (strictly increasing because $\overline{X}_A$ and $\overline{X}_B$ are in normal form) to form two n-tuples ($n=|\overline{X}_A|$):

$$(C(A,X_1),\ C(A,X_2),\ \ldots,\ C(A,X_n))$$

$$(C(B,X_1'),\ C(B,X_2'),\ \ldots,\ C(B,X_n'))$$

Because $\overline{X}_A = \overline{X}_B$ we know that:

$$(\forall X_i \in \overline{X}_A)\ (\exists X_j' \in \overline{X}_B)\ [X_i = X_j']$$

$$(\forall X_i' \in \overline{X}_B)\ (\exists X_j \in \overline{X}_A)\ [X_i' = X_j]$$

Then A and B are homomorphic on X iff $(\forall 1 \leq i \leq n)[X_i = X_i']$

The main property of the homomorphic relation, that the order of complexity is preserved in homomorphic programs, is shown in the following theorem:

Theorem 3: A and B are homomorphic on X iff $(\forall x_1, x_2 \in X)$

$$[x_1\ \underline{relop}_A\ x_2 \Leftrightarrow x_1\ \underline{relop}_B\ x_2]$$

where relop $\in \{>,=,<\}$ and is the same in both cases.

Proof: (=>) Let $x_1 \epsilon X_i$, $x_2 \epsilon X_j$. Suppose $C(A,X_i)$ is in the i^{th} position of its ordered n-tuple (as above) and $C(A,X_j)$ is in the j^{th} position. By homomorphism, $C(B,X_i)$ and $C(B,X_j)$ are in the i^{th} and j^{th} positions of their n-tuple. Then

$$C(A,X_i) \underline{\text{relop}} C(A,X_j) \text{ iff } C(B,X_i) \underline{\text{relop}} C(B,X_j)$$

$$C(A,x_1) \underline{\text{relop}} C(A,x_2) \text{ iff } C(B,x_1) \underline{\text{relop}} C(B,x_2)$$

$$x_1 \underline{\text{relop}}_A x_2 \text{ iff } x_1 \underline{\text{relop}}_B x_2$$

where relop is the same operator in all cases.

(<=) By hypothesis, $(\forall x_1, x_2 \epsilon X)[x_1 \underline{\text{relop}}_A x_2$ iff $x_1 \underline{\text{relop}}_B x_2]$. In particular, $x_1 =_A x_2$ iff $x_1 =_B x_2$, so by theorem 1, A and B are similar on X. Now assume that in the ordered n-tuples for $C(A,X_i)$ and $C(B,X_i')$, $(\exists i)[X_i \neq X_i']$. Let i be the smallest index for which this is true. Suppose the element corresponding to $C(A,X_i)$ is $C(B,X_j)$. We then have:

$$(C(A,X_1), C(A,X_2), \ldots, C(A,X_i) \ldots)$$

$$(C(B,X_1), C(B,X_2), \ldots, C(B,X_j) \ldots)$$

Since i is the smallest index, then X_j could not have occurred within the first i-1 complexities $C(A,X_k)$, $k \leq i-1$. Therefore, $C(A,X_j) > C(A,X_i)$. Similarly, $C(B,X_i) > C(B,X_j)$. If $x_1 \epsilon X_i$, $x_2 \epsilon X_j$, $x_1 <_A x_2$ but $x_1 >_B x_2$. Contradiction! Then our assumption that $X_i \neq X_i'$ was wrong. Thus A and B are homomorphic on X. QED

The homomorphism relation gives us the information we

need to determine the relative magnitudes of the equivalence class complexities of two programs. To see if homomorphism holds, we need only examine the two sets of complexity equivalence classes and the associated complexities. If $|\overline{X}_A| \ll |X|$, the number of objects we will have to examine is small compared to the total number of inputs.

As with similarity, homomorphism is preserved by subset and intersection of input sets:

Theorem 4: If A and B are homomorphic on X and on Y, they are homomorphic on $Z \subseteq X$ and on $X \cap Y$.

Proof: (Subset) By theorem 2, A and B are similar on Z. We can order the equivalence class complexities of A and B on Z:

$$(C(A,Z_1),\ C(A,Z_2),\ \ldots,\ C(A,Z_n))$$

$$(C(B,Z_1'),\ C(B,Z_2'),\ \ldots,\ C(B,Z_n'))$$

where $Z_i = X_i \cap Z$ and $Z_i' = X_i' \cap Z$ as in the proof of theorem 2. Since $X_i = X_i'$, then $Z_i = Z_i'$, $\forall Z_i \in \overline{Z}_A (= \overline{Z}_B)$ $\therefore$ A and B are homomorphic on Z.

(Intersection) $X \cap Y \subseteq X$. Using the first part of this proof, A and B are homomorphic on $X \cap Y$. QED

Homomorphism and Complexity Limits

In Chapter III, we introduced the notion of maximum and minimum complexity on a set of inputs. We noted that these quantities bound the resource usage of a program on a particular input set. Here, we formally define these two quantities and also two others.

Definition: Given a program A with input set X and $\overline{X}_A$ in normal form, we can define the following quantities:

$$C_{max}(A,X) = \max_{X_i \in \overline{X}_A} \{ C(A,X_i) \}$$

$$C_{min}(A,X) = \min_{X_i \in \overline{X}_A} \{ C(A,X_i) \}$$

$$X_{max}(A) = \text{that } X_i \text{ for which } C(A,X_i) = C_{max}(A,X)$$

$$X_{min}(A) = \text{that } X_i \text{ for which } C(A,X_i) = C_{min}(A,X)$$

X_{max} and X_{min} are unique by the normal form condition. They tell us which inputs will consume the greatest amount of the resource being measured and which inputs will consume the least. These two quantities are preserved by homomorphism:

Lemma 4: If A and B are homomorphic on X, then

$$X_{max}(A) = X_{max}(B) \text{ and } X_{min}(A) = X_{min}(B)$$

Proof: Suppose $X_{min}(A) = X_i$. Then

$$(\forall X_j \in \overline{X}_A,\ X_j \neq X_i)[X_i <_A X_j]$$

But by homomorphism,

$$(\forall X_j \in \overline{X}_B,\ X_j \neq X_i)[X_i <_B X_j]$$

$$\therefore\ X_{min}(B) = X_i$$

Similarly, for $X_{max}(A) = X_{max}(B)$. QED

We also state, without proof, the following lemma which relates complexity limits to subsets of input sets:

Lemma 5: If $Z \subseteq X$, then $C_{max}(A,Z) \leq C_{max}(A,X)$ and $C_{min}(A,Z) \geq C_{min}(A,X)$.

Thus C_{max} and C_{min} for the maximal valid input set limit the corresponding quantities for all other valid input sets.

Absolute Homomorphism

Analagous to absolute similarity, we can define two programs to be absolutely homomorphic if they are homomorphic on all valid input sets. Results corresponding to lemmas 2 and 3 can be shown for absolute homomorphism.

Isomorphism

We now define one last equivalence relation between programs with the same input set; this relation is stronger than homomorphism.

Definition: A and B are isomorphic on X iff they are similar on X and

$$(\forall X_i \in \overline{X}_A)[C(A,X_i) = C(B,X_i)]$$

Isomorphism is a special case of homomorphism: the complexity associated with any equivalence class is the same for both programs. As we would expect, isomorphism is preserved by subset and intersection of input sets; however, it is also preserved by union of input sets.

Theorem 5: If A and B are isomorphic on X and on Y, they are also isomorphic on $Z \subseteq X$, $X \cap Y$, and $X \cup Y$.

Proof: (Subset) By theorem 4, A and B are homomorphic on Z.

$$(\forall Z_i \in \overline{Z}_A)\ [C(A,Z_i) = C(A,X_i) = C(B,X_i) = C(B,Z_i)]$$

(Intersection) Follows from subset proof.

(Union) Let $Z = X \cup Y$. First we must show that $\bar{Z}_A = \bar{Z}_B$

$$\bar{Z}_A = \{ X_i \in \bar{X}_A \mid (\forall Y_j \in \bar{Y}_A)[X_i \neq_A Y_j] \}$$

$$\cup \{ Y_i \in \bar{Y}_A \mid (\forall X_j \in \bar{X}_A)[Y_i \neq_A X_j] \}$$

$$\cup \{ X_i \cup Y_j \mid X_i =_A Y_j \}$$

$$= \bar{Z}_{A1} \cup \bar{Z}_{A2} \cup \bar{Z}_{A3}$$

Similarly, $\bar{Z}_B = \bar{Z}_{B1} \cup \bar{Z}_{B2} \cup \bar{Z}_{B3}$. By hypothesis, $\bar{X}_A = \bar{X}_B$, $\bar{Y}_A = \bar{Y}_B$ (all in normal form), and

$$(\forall X_i, Y_j)[C(A,X_i) = C(B,X_i) \ \& \ C(A,Y_j) = C(B,Y_j)$$

$$\therefore \ \bar{Z}_{A1} = \bar{Z}_{B1} \text{ and } \bar{Z}_{A2} = \bar{Z}_{B2}$$

Since $X_i =_A Y_j \iff X_i =_B Y_j$, $\bar{Z}_{A3} = \bar{Z}_{B3}$

$$\therefore \ \bar{Z}_A = \bar{Z}_B$$

$$C(A,Z_i) = \begin{cases} C(A,X_i), & Z_i = X_i \\ C(A,Y_i), & Z_i = Y_i \end{cases}$$

$$C(B,Z_i) = \begin{cases} C(B,X_i), & Z_i = X_i \\ C(B,Y_i), & Z_i = Y_i \end{cases}$$

(Note: if $Z_i = X_i \cup Y_i$, $C(A,X_i) = C(A,Y_i)$ and $C(B,X_i) = C(B,Y_i)$)

But by hypothesis, $C(A,X_i) = C(B,X_i)$ and $C(A,Y_i) = C(B,Y_i)$
Therefore, $C(A,Z_i) = C(B,Z_i)$, $\forall Z_i \in \overline{Z}_A$. Thus A and B are isomorphic on $X \cup Y$. QED

Isomorphism not only preserves X_{max} and X_{min} but also C_{max}, C_{min}, and $C(A,X)$; we state the following easy lemma without proof.

Lemma 6: If A and B are isomorphic on X, then

$$X_{max}(A) = X_{max}(B) \qquad C_{max}(A,X) = C_{max}(B,X)$$

$$X_{min}(A) = X_{min}(B) \qquad C_{min}(A,X) = C_{min}(B,X)$$

$C(A,X) = C(B,X)$ if the methods used in selecting the inputs to A and B (i.e., $Pr(X_i)$) are the same.

We will return to isomorphism in the next section when we discuss concatenation of programs.

4. Concatenation

We will now investigate the complexity properties of programs which may be combined or concatenated to from larger programs. By the concatenation of two programs A and B, we mean the program which is formed by appending B to the end of A,

so that when the locus of execution reaches the end of A, it will enter B. We will assume that A places the outputs of its computation into certain registers and that these same registers are used by B as inputs. B is not requred to use all of A's outputs as inputs: however, B cannot need more inputs than A can supply. We will say that A is I/O-compatible to B if this restriction is obeyed.

We will also assume that neither A or B is changed by the concatenation of B onto A (which will be denoted by "A•B"). We run into at least one trouble spot: If A finishes execution by halting somewhere in the middle. To continue with the execution of B, we would need to change this HALT instruction into a BRANCH to the end of A. This modification would probably change A's complexity with respect to certain inputs. To avoid this difficulty, we will assume that all programs terminate by executing their last instruction. Thus we may perform concatenation without changing instructions in either program.

If we wish to study the complexity of A•B for an input set X, the inputs to B will have to be valid halting inputs. Therefore, A(X), the set of inputs to B, will always be a valid input set. Given this fact, the complexity of A•B for any input x can be determined by linearity:

$$C(A\bullet B,x) = C(A,x) + C(B,A(x))$$

Compatibility

We now place another restriction on the concatenation of programs which will enable us to prove some properties on the complexity of concatenation. This restriction will be defined in two stages as follows:

Definition: Let A be I/O-compatible to B and for X a valid input set to A, A(X) = Y is a valid input set to B. Then A is X_i - compatible to B iff for $X_i \in \overline{X}_A$ (in normal form), there exists $Y_j \in \overline{Y}_B$ (in normal form) such that $A(X_i) = F_i \subseteq Y_j$.

X_i-compatibility tells us that a set of inputs (X_i) which, by definition have the same complexity for A ($C(A,X_i)$) will be transformed into a set of inputs (F_i) for B which will all have the identical complexity - $C(B,Y_j)$. We now extend this relation to all equivalence classes of X:

Definition: A is X-compatible to B iff A is X_i-compatible to B for all $X_i \in \overline{X}_A$ (in normal form).

If A is X-compatible to B, it is easy to see that for $x_1, x_2 \in X$,

$$x_1 =_A x_2 \Rightarrow x_1 =_{A \cdot B} x_2$$

However the converse is not necessarily true. If $x_1 \neq_A x_2$ and $A(x_1) \neq_B A(x_2)$, it may still be that $x_1 =_{A \cdot B} x_2$. By theorem 1,

we conclude that A and A•B are not necessarily similar on X.

Compatibility is preserved by several operations on input sets. We state the following two theorems without including the proofs.

Theorem 6: If A is X-compatible to B and also Y-compatible to B, and if $Z \subseteq X$, A is Z-compatible and $X \cap$ Y-compatible to B.

Theorem 7: If A is X-compatible to B and also Z-compatible to B, and if for $A(X_i) \subseteq Y_i$ and $A(Z_i) \subseteq T_i$, $X_i =_A Z_i \Rightarrow Y_i =_B T_i$, then A is $X \cup Z$-compatible to B.

In the case of theorem 7, we need the additional restriction that $(X_i =_A Z_i \Rightarrow Y_i =_B T_i)$ because if two equivalence classes (X_i, Z_i) must be combined in the normal form of $\overline{(X \cup Z)}_A$ as a result of having the same complexity $(X_i =_A T_i)$, the images (Y_i, T_i) of these classes under A must also have the same complexity $(Y_i =_B T_i)$ so that $A(X_i \cup Z_i) \subseteq Y_i \cup T_i$.

Compatibility and Isomorphism

To conclude the section on concatenation, we discuss some relationships between compatibility and isomorphism. We would like to see under what conditions isomorphic programs can be concatenated to yield programs which are still isomorphic. We first show that isomorphism is preserved by the concatenation of the isomorphic programs to a program which is compatible to

both:

Theorem 8: If A is Y-compatible to B and to C, and if B and C are isomorphic on $X = A(Y)$, then $A \cdot B$ and $A \cdot C$ are isomorphic on Y.

Proof: Let $D = A \cdot B$, $E = A \cdot C$. We need to show that $\bar{Y}_D = \bar{Y}_E$ and $(\forall Y_i \in \bar{Y}_D)[C(D,Y_i) = C(E,Y_i)]$. $\bar{Y}_D$ is composed of two subsets:

$$\bar{Y}_D = \{ Y_i \in \bar{Y}_A \mid (\forall Y_j \in \bar{Y}_A)[Y_i \neq_D Y_j] \}$$

$$\cup \{ Y_j \cup Y_k \mid Y_j =_D Y_k \} = \bar{Y}_{D1} \cup \bar{Y}_{D2}$$

Similarly, $\bar{Y}_E = \bar{Y}_{E1} \cup \bar{Y}_{E2}$

Since $C(B,X_i) = C(C,X_i)$, $\forall X_i \in \bar{X}_B$, it must be that $\bar{Y}_{D1} = \bar{Y}_{E1}$ and $\bar{Y}_{D2} = \bar{Y}_{E2}$. Therefore, $\bar{Y}_D = \bar{Y}_E$

Then $(\forall Y_i \in \bar{Y}_D)$,

$C(D,Y_i) =$ (by linearity)

$C(A,Y_i) + C(B,A(Y_i)) =$ (by isomorphism)

$C(A,Y_i) + C(C,A(Y_i)) =$ (by linearity)

$C(E,Y_i)$

$\therefore$ $A \cdot B$ and $A \cdot C$ are isomorphic on Y. QED

Since A•B and A•C are isomorphic, we can continue the concatenation if we can find a program D which is Z-compatible to A•B and A•C and where $D(Z) \subseteq Y$. Then D•A•B and D•A•C will be isomorphic on Z.

We now show the conditions under which the concatenation of pairs of isomorphic programs results in programs which are also isomorphic.

Theorem 9: Suppose A and B are isomorphic on X, A is X-compatible to C, B is X-compatible to D, C and D are isomorphic on $Y = A(X) \cup B(X)$. Then A•C and B•D are isomorphic on X iff

$$(\forall X_i \in \overline{X}_A)(\exists Y_j \in \overline{Y}_C)[A(X_i),\ B(X_i) \subseteq Y_j]$$

Proof: ($\Leftarrow$) Let E = A•C, F = B•D. Then

$$\overline{X}_E = \{\ X_i \in \overline{X}_A \mid (\forall X_j \in \overline{X}_A)[X_i \neq_E X_j]\ \}$$
$$\cup\ \{\ X_i \cup X_j \mid X_i =_E X_j\ \} = \overline{X}_{E1} \cup \overline{X}_{E2}$$

Similarly, $\overline{X}_F = \overline{X}_{F1} \cup \overline{X}_{F2}$

Since $A(X_i),\ B(X_i) \subseteq Y_j$ and C and D are isomorphic on Y, $C(C,A(X_i)) = C(C,Y_j) = C(D,Y_j) = C(D,B(X_i))$. Also, $C(A,X_i) = C(B,X_i)$, $\forall X_i \in \overline{X}_A$. Thus, $C(E,X_i) = C(F,X_i)$, $\forall X_i$.

$\therefore\ \overline{X}_{E1} = \overline{X}_{F1},\ \overline{X}_{E2} = \overline{X}_{F2}$ and thus $\overline{X}_E = \overline{X}_F$

Then $(\forall X_i \in \overline{X}_E)$,

$C(E,X_i) =$ (by linearity)

$C(A,X_i) + C(C,A(X_i)) =$ (by isomorphism)

$C(B,X_i) + C(C,A(X_i)) =$ (by hypothesis)

$C(B,X_i) + C(C,Y_j) =$ (by isomorphism)

$C(B,X_i) + C(D,Y_j) =$ (by hypothesis)

$C(B,X_i) + C(D,B(X_i)) =$ (by linearity)

$C(F,X_i)$

$\therefore$ A•C and B•D are isomorphic on X.

(=>) Given $A(X_i) \subseteq Y_j$, $B(X_i) \subseteq Y_k$, we want to show that $Y_j = Y_k$. For $X_i \in \overline{X}_A$, $C(E,X_i) = C(F,X_i)$. By linearity, $C(A,X_i) + C(C,A(X_i)) = C(B,X_i) + C(D,B(X_i))$. But $C(A,X_i) = C(B,X_i)$ so we must have that $C(C,A(X_i)) = C(D,B(X_i))$; or $C(C,Y_j) = C(D,Y_k)$. But since C and D are isomorphic on Y, then $\overline{Y}_C = \overline{Y}_D$ (in normal form). Therefore, $Y_j = Y_k$. QED

5. Functional Equivalence

It is often the case that we wish to examine two programs which represent different algorithms for the same function. The programs compute the same output when given the same input from a specified input set; however, the method

(algorithm) used to compute the result is not the same. This situation often arises when we wish to determine which version of a particular subroutine or program we should use. All versions represent (hopefully!) the same function, but one version may use less resources than another. Program equivalence may also arise in the area of simulation. If one program is simulating another, and if the first program is also producing the same output as the second, then the programs are equivalent.

Equivalence is defined here in terms of similar functional behavior on a specified input set. Output values are defined as transformations on input values. If output variables are not also inputs, their values must be expressable in terms of the inputs.

Definition: A and B are equivalent on X iff

$(\forall x \in X)[A(x) = B(x)]$

It is easy to see that equivalence is closed under subset, union, and intersection: if A and B are equivalent on X and on Y, they are equivalent on $Z \subseteq X$, $X \cap Y$, and $X \cup Y$. Equivalence and concatenation are related by the following lemma:

Lemma 7: If A and B are equivalent on X, C and D equivalent on Y where $A(X) \subseteq Y$; and if A, B are I/O-compatible to C, D then A•C, A•D, B•C, B•D are equivalent on X.

Proof: By I/O-compatibility restriction, A•C, etc. may be correctly formed. Now for $x \in X$, $A(x) = B(x) = y$. Since $y \in Y$, $C(y) = D(y)$. Therefore, $A \cdot C(x) = B \cdot D(x)$ and A•C and B•D are equivalent on X. Similarly for the other cases. QED

To conclude this chapter, we present the following theorem which relates isomorphism, compatibility and equivalence:

Theorem 10: Suppose A and B are isomorphic and equivalent on X, A is X-compatible to C, B is X-compatible to D, C and D isomorphic on $Y = A(X) = B(X)$ (since A and B equivalent). Then A•C and B•D are isomorphic on X.

Proof: We look at theorem 9. Since $A(x) = B(x)$, $\forall x \in X$, $A(X_i) = B(X_i)$, $\forall X_i \in \bar{X}_A$. Since A is X-compatible to C, $A(X_i) \subseteq Y_j$, $Y_j \in \bar{Y}_C$. But by the isomorphism of C and D on Y, $Y_j \in \bar{Y}_D$. Then $B(X_i) \subseteq Y_j$. Therefore, A•C and B•D are isomorphic on X. QED

If C and D are equivalent on Y in addition to being isomorphic, we can use lemma 7 to conclude that A•C and B•D are isomorphic and equivalent on X.

Chapter VI. Conclusions and Suggestions for Further Study

Conclusions

We have developed a general theory of computational complexity for computer programs. We have looked at complexity from the viewpoint of resource usage and regarded the use of different resources as different measures of the complexity of a program. Many complexity measures fit into our theory, the only requirement being that the usage of the associated resource obey the linearity principle.

Our theory has been based upon observing the behavior of a program on a valid set of inputs. We have also relied extensively on an equation representation of the transformations which a program applies to its inputs. We have attempted to justify the use of finite input sets by noting that real programs, running on real computers, are able to accept and manipulate only a finite number of distinct input values, due to software and hardware limitations. In addition, most programs actually process only a subset of all possible input values. Our methods were shown to be valid for programs with a large number of programming linguistic constructs and with several different input schemes.

The complexity analysis of a program produced a set of complexity equivalence classes and a corresponding set of class complexities. We have noted that most programs do not

treat every input value differently, and therefore the number of equivalence classes will generally be smaller than the number of possible inputs. Thus, these two sets give a relatively compact representation of the complexity information for a program operating on a given input set. These sets immediately led to some complexity parameters for the program:

1. $C(A,X)$ - the expected complexity value on the set X
2. $C_{max}(A,X)$ - the maximum complexity on the set X
3. $X_{max}(A)$ - those inputs which result in complexity $C_{max}(A,X)$
4. $C_{min}(A,X)$ - the minimum complexity on the set X
5. $X_{min}(A)$ - those inputs which result in complexity $C_{min}(A,X)$

$C_{max}(A,X)$ and $C_{min}(A,X)$ bound the resource usage of program A and tell how much of this resource A can possibly use and how much it must use to process inputs from set X.

Our basic purpose has been to develop some theoretical tools for studying the difficulty of computing functions by observing the program implementations of these functions. While these techniques will work for any program of the types described, the necessary computations will become unmanageable if the program equation is complex or if the number of complexity equivalence classes is large. Thus, one could not expect to sit down and find the equation and equivalence classes of a PL/I compiler, just as one would be hard-pressed to prove that such a compiler is "correct". Our techniques will probably be most valuable in the analysis of small programs and also

in deciding which version of a particular program will be most suitable for use; suitable in the sense of using the least resources over a particular set of inputs, where we may wish to minimize $C(A,X)$, $C_{max}(A,X)$, or $C_{min}(A,X)$.

Programmers and systems analysts are often faced with this latter problem, particularly in a large programming system such as a language translator or operating system which is modularized and which has its basic components frequently replaced as the system is up-graded. The complexity analysis techniques allow different versions of program modules to be compared, perhaps in one case on the basis of maximum resource usage, in another with regard to average resource usage, in a third with respect to some combination of complexity parameters.

Areas for Further Study

We have used the concept of a program equation for much of our work. This equation is independent of a study of complexity. It gives a concise algebraic formulation of a program or algorithm. It is also valid on an infinite input set as long as all elements in this set are halting inputs for the program in question. The equation brings to light the transformational characteristics of a program. It should be useful in the study of other aspects of programs and programming languages, such as program semantics and program correctness.

A subject which we have not examined but which is important is the use of space as a complexity measure. The use of space did not follow the linearity principle. An analagous constraint for space and its related measures would have to be devised to analyze the complexity of programs with respect to these resources.

We have not discussed the effects of transformations of the program on the complexity. For example, suppose we make a well-defined modification to program A with input set X. Is there a well-defined effect on $\overline{X}_A$, $C_{max}(A,X)$, $C_{min}(A,X)$? Cooper's work on graph transformations [8] may be useful here.

We have defined three equivalence relations between programs with the same input set. There may be other relations, intermediate in strength between similarity and isomorphism which reflect other programming situations.

We have mentioned that it is advantageous for $|X| \gg |\overline{X}_A|$. In this case, $\overline{X}_A$ and $\{ C(A,X_i) \mid X_i \in \overline{X}_A \}$ give us a more economical representation of the complexity information for A than if we simply looked at all x in X. However, we have not explored the relationship between the relative size of $\overline{X}_A$ and the nature of A itself. Are there certain conditions for which we get this economical representation?

Appendix - Mathematical Notation

Sets

We make use of the notion of a set and various operations upon it. A set is an unordered collection of objects and is named by a capital letter. The elements of a set are enclosed in braces and separated by commas.

A subset of a given set is another set containing all, some, or none of the elements of the original set. If Z is a subset of X, we write $Z \subseteq X$. If a subset contains no elements, it is called the empty set and denoted by ϕ.

The size of a set X is written $|X|$ and is simply the number of elements in X.

The union of the sets X and Y, denoted $X \cup Y$, is a set containing those elements which are in either X or Y or both.

The intersection of sets X and Y, denoted $X \cap Y$, is a set containing those elements which are in both X and Y.

We denote membership in a set by the symbol $\in$. Thus, $a \in \{ a,b \}$. Similarly, $c \notin \{ a,b \}$.

A set may be specified by describing the conditions for membership in the set rather than listing all of the members. We use the notation

$$X = \{ x \mid \text{"conditions"} \}$$

This may be read "X is the set of all objects x such that the

conditions are true". Thus the set

$$Y = \{ x \mid x \epsilon X_1 \text{ and } x \epsilon X_2 \}$$

describes Y as the intersection of X_1 and X_2.

Ordered Pairs, Cross Products

An ordered pair, denoted (x,y), is an object with two components. The order of the components is significant: (x,y) is distinct from (y,x) unless x = y.

A set of ordered pairs can be formed from sets of single objects by the cross product operator. The cross product of the sets X and Y, denoted XxY, is defined as follows:

$$X \times Y = \{ (x,y) \mid x \epsilon X \text{ and } y \epsilon Y \}$$

Quantifiers

We use two logical quantifiers in our notation. The universal quantifier, denoted $\forall$, may be read "for all". It is used to qualify the statement following it. Thus,

$$(\forall x \epsilon X)]x \leq y]$$

means "for all x in X, $x \leq y$".

The existential quantifier, $\exists$, may be read "there exists". The statement

$$(\exists x \epsilon X)[x \leq y]$$

means "there exists an x in X such that $x \leq y$".

Quantifiers may be grouped in a series to form more complex logical statements. We might have

$$(\forall x \in X)(\exists y \in Y)[x \leq y]$$

which states that "for all x in X, there exists a y in Y such that $x \leq y$".

Functional Composition

The composition of functions f and g, denoted $f \cdot g$, is another function which transforms the domain of g into the range of f. Thus if $g(x) = y$ and $f(y) = z$, $f \cdot g(x) = z$. Composition may be continued for any number of functions. If a function f is composed with itself i times, we may abbreviate this as f^i.

Implication and Equivalence

If the truth of statement A implies the truth of statement B, we write

$$A \Rightarrow B$$

This may also be read "if A then B". Similarly if B implies A, we write

$$B \Rightarrow A$$

If A and B imply each other, they are said to be equivalent and we write

$$A \equiv B$$

We can also write equivalence as

$$A \text{ iff } B$$

where "iff" is an abbreviation for "if and only if".

References

1. Hartmanis, J., and J. E. Hopcroft, "An Overview of the Theory of Computational Complexity", Technical Report No. 70-59, Department of Computer Science, Cornell University, April 1970. (See also JACM 18(3), July 1971, for a later version of this report).

2. Landin, P. J., "The Mechanical Evaluation of Expressions", Computer Journal 6, 4 (January 1964) pp. 308-320.

3. Blum, M., "A Machine Independent Theory of the Complexity of Recursive Functions", JACM 14 (1967) pp. 322-336.

4. Meyer, A. R., and D. M. Ritchie, "The Complexity of LOOP Programs", Proceedings of the 22nd National ACM Conference, (1967), pp. 465-469.

5. Ramamoorthy, C. V., "Discrete Markov Analysis of Computer Programs", Proceedings of 20th National ACM Conference, (1965), pp. 386-392.

6. Zeiger, H. P., "Formal Models of Some Features of Programming Languages", Proceedings of the 3rd Annual Princeton Conference on Information Science and Systems, (1969), pp. 425-429.

7. Drake, A., Fundamentals of Applied Probability Theory, McGraw-Hill Book Company (1967), New York.

8. Cooper, D. C., "Some Transformations and Standard Forms of Graphs, with Applications to Computer Programs", in Machine Intelligence 2, Dale and Michie (ed.) American Elsevier Publishing Company, Inc., (1968), New York.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Massachusetts Institute of Technology Project MAC	UNCLASSIFIED
	2b. GROUP: None

3. REPORT TITLE

Complexity Measures for Programming Languages

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Memorandum

5. AUTHOR(S) (Last name, first name, initial)

Goodman, Leonard I.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
September 1971	86	8

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Nonr-4102(01)	TM-17
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

A theory of complexity is developed for algorithms implemented in typical programming languages. The complexity of a program may be interpreted in many ways; a method for measuring a specific type of complexity is a complexity measure -- some function of the amount of a particular resource used by a program in processing an input.

After the complexity of the basic program elements is determined, program complexity is analyzed with respect to single inputs and then with respect to finite sets of legitimate halting inputs. A program equation is developed to aid in the complexity analysis. Using this equation, an input set is partitioned into classes of constant complexity.

Several equivalence relations are defined, relating different programs by their complexity. Complexity is also discussed in terms of concatenation and functional equivalence of program.

14. KEY WORDS

Computational Complexity	Complexity Measures	Program Equations
Program Resource Usage	Programming Languages	Program Equivalence Relations

DD FORM 1 NOV 68 1473 (M.I.T.)