A STRUCTURED MODEL OF PROGRAMS SUITABLE FOR ANALYSING TIME/STORAGE TRADE-OFFS

Anthony G. Middleton

Department of Computer Science,
University College of Swansea,
Swansea, SA2 8PP, Wales, UK.

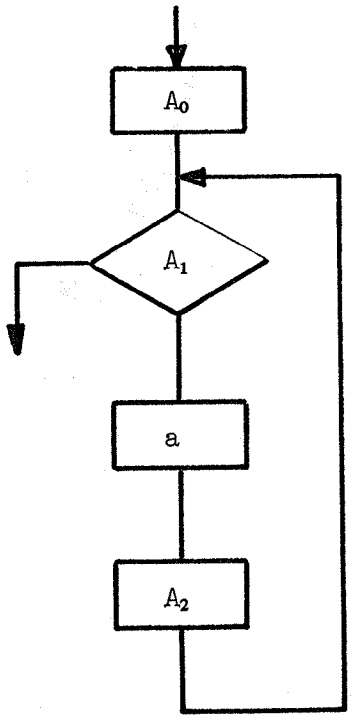
1.0 SUMMARY

A program is modelled as a graph whose nodes (actions) are characterised by symbols for their time and storage requirements. The graph is expressed as a hierarchy of basic graphs (constructs). Each basic construct has characteristic equations for its time and storage requirements (resource equations). Recursive application of these basic resource equations allows resource equations to be derived for any program represented in this manner. Furthermore, certain basic constructs have alternative, equivalent constructs. Such design decisions may occur at several points in the program, giving rise to a set of equivalent "variants" of the program. Using such a model, this set of equivalent variants can be systematically enumerated, resource equations can be derived for each variant, and, given a meaningful cost equation, a minimum cost variant of the program can be selected. Symbols have to be supplied for the resource requirements of the atomic (unstructured) actions and for relevant cycle counts and branching probabilities. In a future system values for these symbols might be derived from automated program measurement.

2.0 THE INITIAL MODEL

We first discuss the simplest presentation of the model: the model is being extended to give it a wider range of applicability. The notion of a "graph function" aids exposition (this notion deriving from [1]). An example of a graph function is:-

DO(N,a) =



The inputs to the graph function can be any one of the following types of item:-

(i) An integer.
(ii) A probability.
(iii) An action.

The output is a program graph. The above graph can be taken as a model of the FORTRAN DO-loop construct with the following interpretations:-

A_0 ∿ initialisation of the control variable
A_1 ∿ testing of the control variable
A_2 ∿ incrementing the control variable
N ∿ the number of complete cycles
a ∿ the action embedded in the DO-loop.

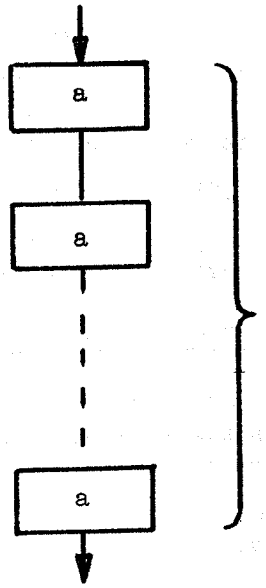
Different versions will exist for different implementations. The time and storage equations for this construct are:-

$$TIME(DO(N,a)) = T_0 + N * (T_1 + TIME(a) + T_2) + T_1$$
$$STORE(DO(N,a)) = S_0 + S_1 + S_2 + STORE(a)$$

where T_1, S_1 are the time and storage requirements of A_1, etc.
(Note that A_1 is performed N+1 times.)

Another graph function is

REPEAT(N,a) =



N occurrences

The resource equations for the REPEAT construct are:-

$$TIME(REPEAT(N,a)) = N * TIME(a)$$
$$STORE(REPEAT(N,a)) = N * STORE(a)$$

which represents N successive occurrences of the action a. In certain cases where a DO-loop is used, a pattern of "repeated code" could be employed instead. e.g.

```
      DO  100 I=1,50
100   X(I) = Y(I) + T
```

(where X and Y are arrays) could be replaced by:-

```
      X(1) = Y(1) + T
      X(2) = Y(2) + T
             .
             .
             .
     X(50) = Y(50) + T
```

The first alternative could be modelled by use of the DO graph function, and the second by REPEAT. This choice of construct can be denoted thus:-

DO(N,a) ≡ REPEAT(N,a) ≙ DOREP(N,a)

Thus an occurrence of DOREP is taken to indicate a choice between the use of a DO-loop and the use of repeated code.

Using this notation, the expression:-

ε = DOREP(M,DOREP(N,b))

can be used to denote a choice from the following four variants:-

DO(M,DO(N,b))
DO(M,REPEAT(N,b))
REPEAT(M,DO(N,b))
REPEAT(M,REPEAT(N,b))

and recursive use of the resource equations for the constructs involved allows the resource equations to be derived for each variant. For example:-

TIME(DO(M,REPEAT(N,b)))
= T_0 + M * (T_1 + TIME(REPEAT(N,b)) + T_2) + T_1

(using time equation for the DO construct)

= T_0 + M * (T_1 + N * TIME(b) + T_2) + T_1

(using time equation for the REPEAT construct).

If an acceptable cost equation, COST, can be found such that:-

COST(TIME(p), STORE(p))

indicates the cost of a program p, then a minimum-cost variant can be selected whenever the values for the atomic symbols are known or can be measured.

3.0 DEFINING GRAPH FUNCTIONS IN TERMS OF PRIMITIVES

Graph functions can be defined in terms of a set of "primitives". The choice of primitives is somewhat arbitrary. The following set of primitives is used to indicate the approach:-

Simple actions
Compound actions
Cycles
Repetitions
Join-branches.

Each primitive has characteristic resource equations and these can be used to derive resource equations for any graph function which is defined in terms of the primitives. .The above primitives are described below.

3.1. Simple Actions

This is an "atomic" action which is simply characterised by symbols or expressions for its resource requirements.

3.2 Compound Actions

COMPOUND(a_1 , ..., a_n) is used to denote the serial composition of actions a_1 , ..., a_n.

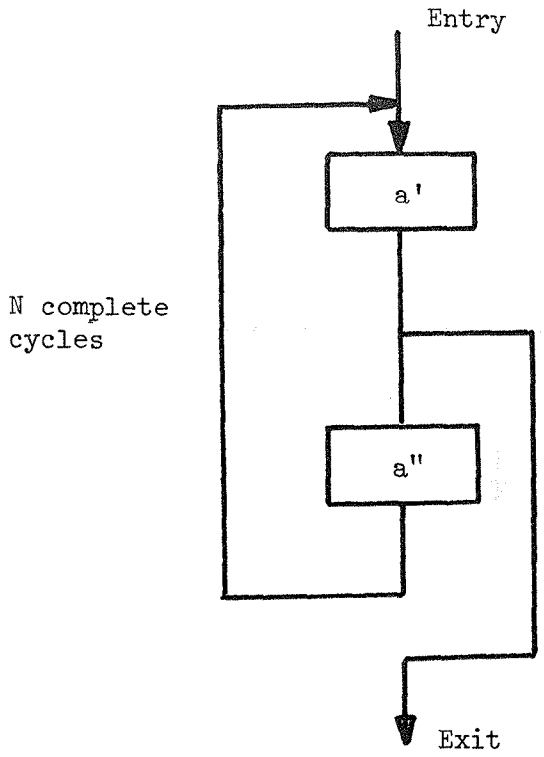
The resource equations are:-

$$TIME(COMPOUND(a_1, \ldots, a_n)) = TIME(a_1) + \ldots + TIME(a_n)$$

$$STORE(COMPOUND(a_1, \ldots a_n)) = STORE(a_1) + \ldots + STORE(a_n)$$

3.3 Cycles

CYCLE(N,a',a") is used to denote the structure:-



If N complete cycles are performed, part of the cycle (a') is performed N+1 times. The resource equations are:-

$$TIME(CYCLE(N,a',a")) = N * (TIME(a') + TIME(a")) + TIME(a')$$

$$STORE(CYCLE(N,a',a")) = STORE(a') + STORE(a")$$

3.4 Repetitions

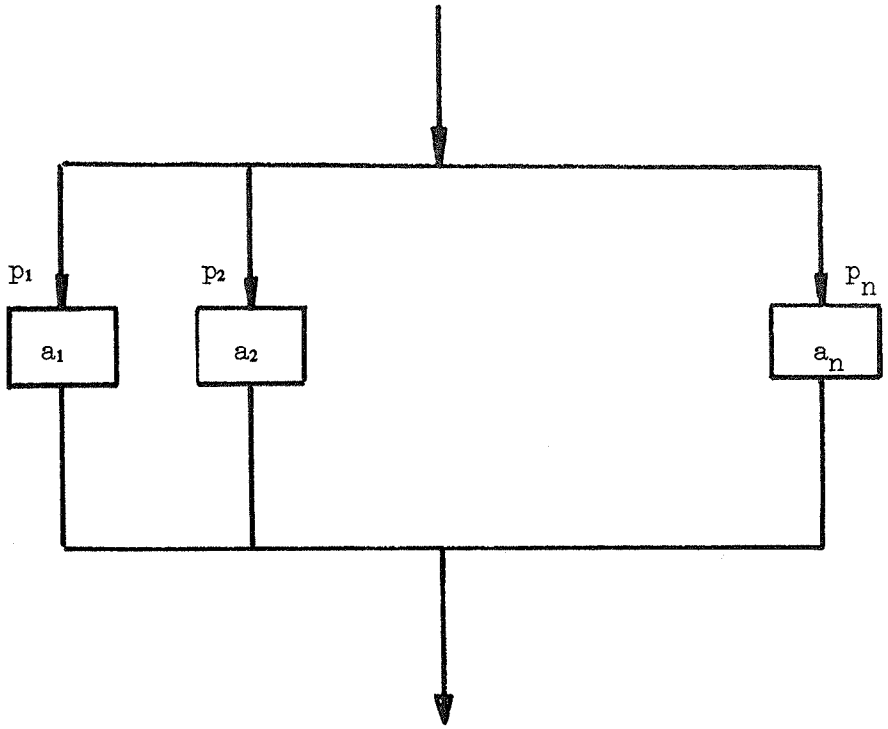
REPEAT(N,a) indicates n sequential occurrences of the action a: possibly with systematic variations of coding which are assumed not to affect the resource requirements of a. The resource equations are:-

$$TIME(REPEAT(N,a)) = N * TIME(a)$$

$$STORE(REPEAT(N,a)) = N * STORE(a)$$

3.5 Join-Branches

JOINBRANCH(⟨ p_1 , a_1 ⟩, ⟨ p_2 , a_2 ⟩, ... ⟨ p_n, a_n⟩)
denotes the structure:-



The action a_1 is performed with probability p_1 , the action a_2 is performed with probability p_2 , etc. The resource equations are:-

$$\text{TIME}(\text{JOINBRANCH}(\langle p_1, a_1\rangle, \langle p_2, a_2\rangle, \ldots, \langle p_n, a_n\rangle)) = p_1 * \text{TIME}(a_1) + p_2 * \text{TIME}(a_2) + \ldots p_n * \text{TIME}(a_n)$$

$$\text{STORE}(\text{JOINBRANCH}(\langle p_1, a_1\rangle, \langle p_2, a_2\rangle, \ldots, \langle p_n, a_n\rangle)) = \text{STORE}(a_1) + \text{STORE}(a_2) + \ldots + \text{STORE}(a_n)$$

The expression for time is a statistical average.

The DO graph function could be expressed in terms of these primitives as:-

```
DO(N,a) = COMPOUND(ACTION(0), CYCLE(N,ACTION(1),
                    COMPOUND(a,ACTION(2))))
```

where ACTION(0) denotes an atomic action whose resource requirements will be represented by symbols T_0 and S_0 , etc.

The resource equations for this graph function can be derived from the resource equations of the primitives:-

```
TIME(DO(N,a)) =

TIME(COMPOUND(ACTION(0),CYCLE(N,ACTION(1),
        COMPOUND(a,ACTION(2)))))

    = TIME(ACTION(0)) +
      TIME(CYCLE(N,ACTION(1),COMPOUND(a,ACTION(2))))

    = T0 + N * (TIME(ACTION(1)) + TIME(COMPOUND(a,ACTION(2)))
        + TIME(ACTION(1))

    = T0 + N * (T1 + TIME(a) + TIME(ACTION(2))) + T1

    = T0 + N * (T1 + TIME(a) + T2 ) + T1
```

4.0 COMMENT ON CURRENT PROGRAM OPTIMISATION TECHNIQUES

If S and T denote the time and store requirements of a program, then current optimisation methods can be regarded as the application of a sequence of equivalence-preserving transformations to a program which, individually, may have one of the following four effects:-

(i) T is reduced, S is reduced (e.g. removal of a redundant action).
(ii) T remains constant, S is reduced (e.g. removal of an unreferenced data structure).
(iii) T is reduced, S remains constant (e.g. movement of some action to a less frequently executed part of the program).
(iv) T is reduced, S may be increased: but the increase is regarded as unimportant. (This is presumably the case in the treatment that would be given to conditionals inside DO-loops in [2]).

In cases (i) to (iii) the advantages appear obvious. However in case (iv), the situation is not so obvious. Using our representation of a DO-loop, case (iv), for moving a conditional outside a DO-loop, can be portrayed as in Figure 1. We could define these two alternatives so:

```
PHI(N,c,p,a,b) =
DO(N,COMPOUND(c,JOINBRANCH(⟨ p,a ⟩ ,⟨ 1-p,b ⟩ )))

RHO(N,c,p,a,b) =
COMPOUND(c,JOINBRANCH(⟨ p,DO(N,a) ⟩ ,⟨ 1-p,DO(N,b)⟩ ))
```

Resource equations can be derived for each alternative and, given values for the primitive symbols, a minimum cost variant chosen - no doubt giving us the same verdict as Schneck and Angel!

However, the situation becomes more complicated if one considers the definitions:-

```
PHI(N,c,p,a,b) =
DOREP(N,COMPOUND(c,JOINBRANCH(⟨ p,a ⟩ ,⟨ 1-p,b ⟩ )))

RHO(N,c,p,a,b) =
COMPOUND(c,JOINBRANCH(⟨ p,DOREP(N,a) ⟩ ,⟨ 1-p,DOREP(N,b)⟩ ))
```

since PRO(N,c,p,a,b) now denotes the set of alternatives:-

```
DO(N,COMPOUND(c,JOINBRANCH(⟨ p,a ⟩ ,⟨ 1-p,b ⟩ )))
REPEAT(N,COMPOUND(c,JOINBRANCH(⟨ p,a ⟩ ,⟨ 1-p,b ⟩ )))
COMPOUND(c,JOINBRANCH(⟨ p,DO(N,a) ⟩ ,⟨ 1-p,DO(N,b)⟩ ))
COMPOUND(c,JOINBRANCH(⟨ p,REPEAT(N,a) ⟩ ,⟨ 1-p,DO(N,b)⟩ ))
COMPOUND(c,JOINBRANCH(⟨ p,DO(N,a) ⟩ ,⟨ 1-p,REPEAT(N,b)⟩ ))
COMPOUND(c,JOINBRANCH(⟨ p,REPEAT(N,a) ⟩ ,⟨ 1-p,REPEAT(N,b)⟩ ))
```

The repeated code option only applies when the upper limit of the DO-loop control variable is known at compile time. Other conditions can give rise to other alternatives to the DO-loop. For example, one might use a "diluted" DO-loop in which the embedded action is performed M times on each cycle of the loop. This alternative could be defined by the graph function (see Figure 2):-

```
DILDO(N,M,a) =
     DO(N/M,REPEAT(M,a))
```

where M divides N and:-

$$TIME(DILDO(N,M,a)) = T_0 + N/M * (T_1 + TIME(a) + T_2) + T_1$$
$$STORE(DILDO(N,M,a)) = S_0 + S_1 + S_2 + M * STORE(a)$$

(this is an approximation - the DO-loop overheads may alter because of the different nature of the operations on the DO-loop parameters. Also, see below, Section 5.2.)

In a situation in which all three versions of the DO-loop are permissible, the optimum variant of PRO(N,c,p,a,b) becomes somewhat non-obvious, and mechanised enumeration of alternatives appears worthwhile.

5.0 SOME EXTENSIONS TO THE MODEL

A "program analyser" which is based on the above model is at present near completion. It allows such program design problems to be posed in a manner which is reasonably natural for a systems programmer. It essentially removes the tedious burden of algebraic manipulation in analysing time/storage trade-offs. At present the system is "spoon-fed" by the systems analyst. But the eventual aim is for the analyser to be resident in a computer and to acquire its information (and implement its decisions) mechanically. Below are some modifications to the model which are currently being studied.

5.1 Allowing for more Resource Types

One could classify resources into two types:-

(i) "Per Node" resources. These resources are associated with the existence of the node (e.g. store).
(ii) "Per Visit" resources. A quantum of the resource is used on each visit to the node (e.g. time).

Resource equations can be found for each resource type and construct e.g. if PN denotes a per node resource, and PV denotes a per visit resource, we have:-

$$PV(CYCLE(N,a',a")) = N * (PV(a') + PV(a")) + PV(a')$$
$$PN(CYCLE(N,a',a")) = PN(a') + PN(a")$$

This approach might allow treatment of program involving input/output: such operations requiring a "per visit" resource.

5.2. Interaction of Constructs

The choice between use of a DO-loop and use of repeated code was presented as a choice between DO(N,a) and REPEAT(N,a). It should more correctly be presented as a choice between DO(N,a) and REPEAT(N,a') where a' is a modified form of a which may require slightly different resources. Thus, resource requirements of an action can be influenced by neighbouring design decisions. This is provided for in the program analyser by employing "conditional actions" whose resource requirements are dependent on a decision variable. If the

resource requirements of an action are influenced by several decisions: the analysis can still be conveniently handled provided that the influences are separable.

5.3 The Influence of Data Structures

Work is in hand to study the influence of data structure design on the resource requirements of a program. It is worth noting in passing that store equations exist for data structures e.g.

STORE(LIST(N,a)) = N * (2 + STORE(a))

might be a suitable equation for the store required for a list of N cells containing data structures of type "a": where a is some complex structure. Such equations would be most useful where all sub-structures of a data structure were of the same class. (One can include probabilities in an obvious manner - but finding values for the probabilities might be another matter.)

5.4 Cost Equations

A useful piece of further research would be to try and find a systematic basis for determining the "cost" of a program. The nature of the cost equation will depend on the nature of the computer configuration.

5.5 Multiple Entries and Exits

At present a program is modelled as a hierarchy of single-entry, single-exit constructs. These could be the nodes of a more arbitrarily structured graph in which the nodes might have several entries and/or several exits. Such an extension is being implemented.

5.6 Complexity of Analysis

If there are N independent, 2-way decisions to be made in designing a program, there are 2^N possible variants of the program and 2^{N+1} resource equations to be derived. Clearly, there is a problem of complexity and some balance must be struck between the complexity of analysis and the benefits derived. One obvious approach is to analyse only those choices occurring in the innermost constructs of the most frequently executed parts of the program.

Other possibilities exist.

5.7 A More Empirical Approach

One may wish to define a construct in the following manner:-

SUB(N_{PA},a):
TIME = $T_3 + T_4 * N_{PA}$ + TIME(a):
STORE = $S_3 + N_{PA}$:

Here, SUB(N_{PA},a) is intended to model the action a embedded in a subroutine requiring N_{PA} parameters. The equations for the resource requirements of a subroutine are given directly - no attempt is given to give a graphical description of the construct (such attempts proved awkward and unnatural). In this model, it is assumed that the subroutine is used at several points in the program and that the storage cost of the single copy of the subroutine can be ignored: other possibilities exist. The alternative to using a subroutine is the "open" coding of a.

This alternative can be expressed by the trivial graph function:-

$$OPEN(N_{PA},a) = a$$

and the choice between the two constructs specified by:-

$$OPEN(N_{PA},a) \equiv SUB(N_{PA},a) \triangleq SUBOP(N_{PA},a)$$

From the above it should be clear that this is very much an on-going piece of work. Problems are in sight - but so also are solutions. It should be noted that the approach is suited to the analysis of sequential systems. No attempt has yet been made to deal with parallelism and pipelining.

As a parting shot, the author suggests that deriving by hand the resource equations of the 24 variants of:-

PRO(N,c,p,SUBOP(N1,a),SUBOP(N2,b))

(a very simple structure!) would illuminate the motivation behind this work.

ACKNOWLEDGEMENTS

I would like to thank Professor D. W. Barron of Southampton University for advice and encouragement throughout this project. I would also like to thank the Science Research Council for financial support.

REFERENCES

[1] Flow Diagrams, Turing Machines and Languages with only two Formation Rules. C. Böhm and G. Jacopini, CACM Vol 9, No. 5, pp. 366-371.

[2] A Fortran to Fortran Optimising Compiler. P. B. Scheck and E. Angel, Computer Journal, Vol. 16, No. 4, pp. 322-329.

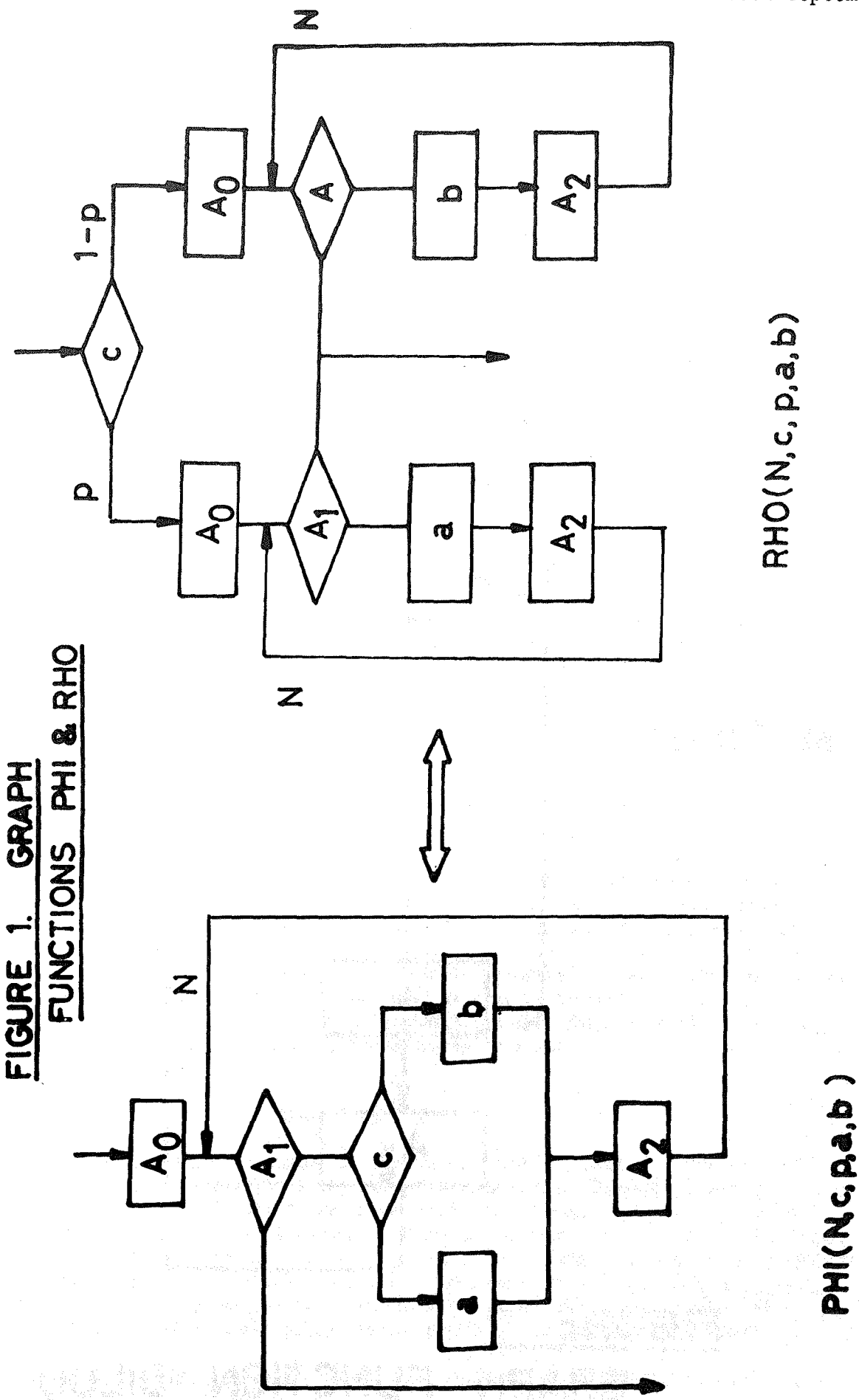


FIGURE 1. GRAPH FUNCTIONS PHI & RHO

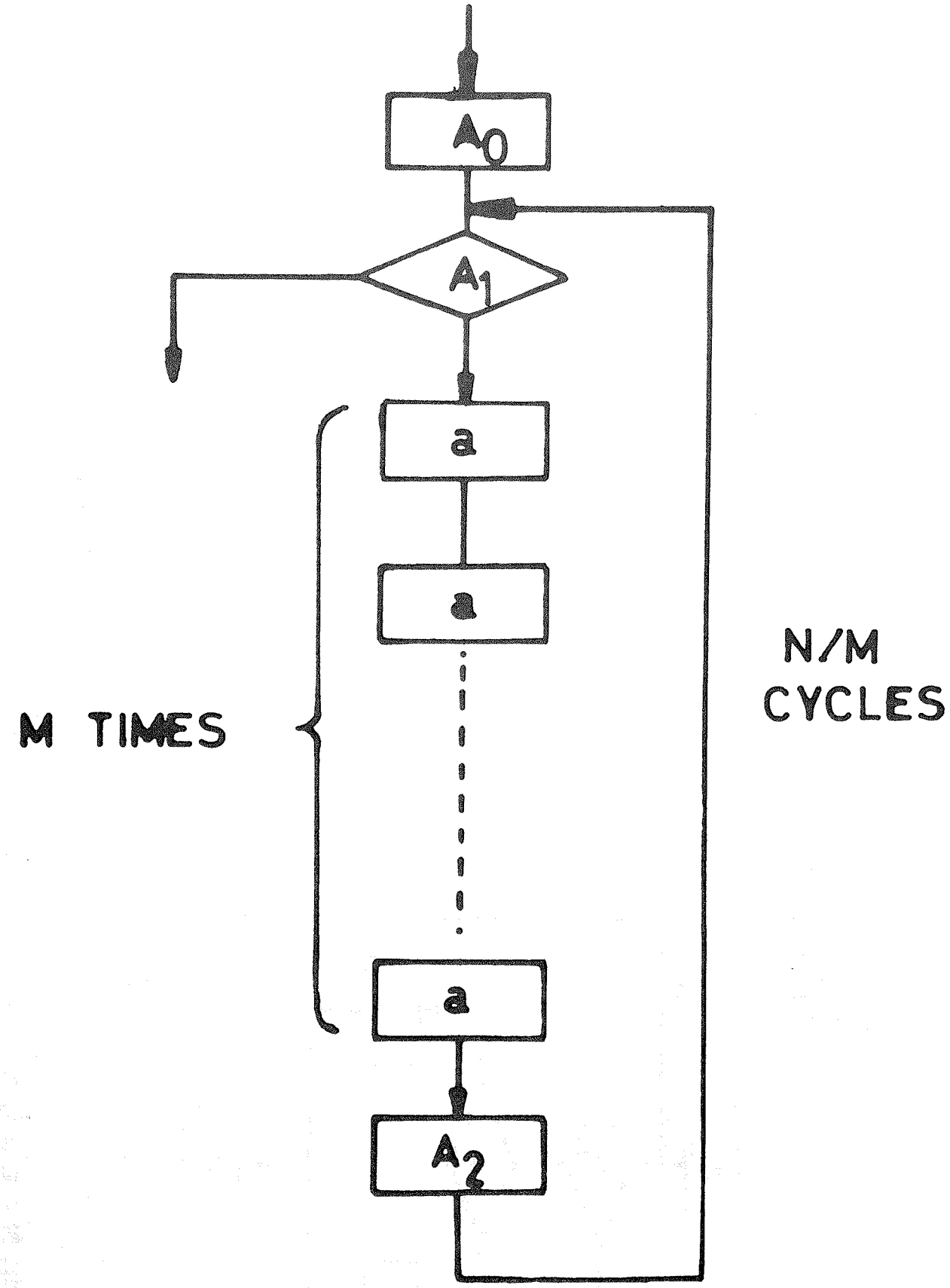


FIGURE 2

GRAPH FUNCTION DILDQ