Applications of Boolean Matrices to the Analysis of Flow Diagrams

REESE T. PROSSER†

Introduction

ANY SERIOUS attempt at automatic programming of large-scale digital computing machines must provide for some sort of analysis of program structure. Questions concerning order of operations, location and disposition of transfers, identification of subroutines, internal consistency, redundancy and equivalence, all involve a knowledge of the structure of the program under study, and must be handled effectively by any automatic programming system.

The structure of a program is usually determined by detailed specifications describing the program, and may usually be given a convenient geometric representation by means of flow diagrams. Ordinarily, neither of these forms is immediately adaptable for handling by machine, and for this purpose another representation of the same information must be found. Such a representation should certainly have these properties:

(1) It should be easy to construct and reproduce.

(2) It should be adaptable to handling by machine.

(3) It should contain all of the information provided by the topology of the flow diagram.

The Connectivity Matrix

A representation which has all these properties may be given by means of Boolean matrices. By a Boolean matrix we mean a matrix whose entries consist entirely of 0's and 1's. The representation is constructed as follows: Suppose we are given the structure of a program, say in the form of a flow diagram consisting of boxes, representing program operations, connected by directed line segments, representing the program flow. We are interested only in the structure, or connectivity, of this diagram, and not in the properties of the individual boxes. We make no restrictions at all on the connectivity and, in particular, branches and loops of all kinds are admissible. We begin by numbering the boxes of the diagram, say from 1 to n, in any convenient manner whatever. For later convenience we adjoin to the diagram a box numbered 0 as the initial, or input, position and a box numbered $n + 1$ as the final, or output, position of the diagram. We next construct an $(n + 2) \times (n + 2)$ Boolean matrix, $A = (a_{ij})$, called the *connectivity matrix* associated with the diagram, by stipulating that $a_{ij} = 1$ if the diagram contains a directed line segment leading directly from box i to box j, and $a_{ij} = 0$ otherwise. Thus $a_{ij} = 1$ if box i may be followed *immediately* by box j in the program, and 0 otherwise.

It is evident that this matrix is easy to construct and easy to handle. It is determined uniquely by the diagram, up to a permutation of the entries due to a renumbering of the boxes, and in turn it determines the diagram, in the sense that the diagram may be completely reconstructed from the matrix. Thus it meets all of our requirements.

This idea is certainly not new. Boolean matrices have been used extensively to study the connectivity and orientation of graphs [7], [12]; networks [4], [6]; organization and group dynamics problems [8]; and more generally, finite Markov processes [11]. Shannon [13] has pointed out that every flow diagram is essentially a finite Markov process, so that we have here a very special case of [11]. On the other hand it is worth emphasizing how well this idea adapts itself to program analysis. A similar attempt with a somewhat different viewpoint appears in [14].

Analysis

Certain elementary computations on the connectivity matrix yield detailed information on the program flow. To show how this comes about, we define a one-row matrix

$$e_i = (0, 0, \ldots, 1, \ldots, 0)$$

with 1 in the ith place and 0's elsewhere. Then, from the definition of A, we see that the matrix product e_iA is a one-row matrix which has 1 in the jth column if it is possible to proceed from box i to box j in one step, and 0 otherwise. By repeating this argument, we see that the *product* $e_iA^2 = (e_iA)A$ is a one-row matrix whose jth column is 1 (or more) if it is possible to proceed from box i to box j in *exactly two steps*, and 0 otherwise. A similar interpretation may evidently be given to higher powers of A.

Now A^2 need not be a Boolean matrix. But it is clear that for our purpose we lose nothing if we replace all *non-zero* entries in A^2 with 1's. This amounts to multiplying A by A according to the following rule: *The Boolean product $A \vee B$ of the Boolean matrices A and B is that Boolean matrix whose i-j entry is*

$$\bigvee_k (a_{ik} \wedge b_{kj})$$

Here $\vee$ and $\wedge$ denote the Boolean operations of max

* The work reported in this paper was performed at Lincoln Laboratory, center for research operated by M.I.T. with the joint support of the U. S. Army, Navy and Air Force.

† Massachusetts Institute of Technology Lincoln Laboratory, Lexington, Mass.

and min, respectively. In the same spirit we define: *The Boolean sum $A \wedge B$ of the Boolean matrices A and B is that matrix whose i-j entry is $a_{ij} \vee b_{ij}$.* Thus Boolean sums and products of Boolean matrices are formed in the same way as ordinary matrix sums and products, except that $+$ is replaced by $\vee$ and $\times$ by $\wedge$.

Now the way is clear for induction. Let A be the connectivity matrix of a flow diagram, and define

$$A_m = A_{m-1} \wedge A = A \wedge A \wedge \ldots\ldots\ldots \wedge A \quad m \text{ times}$$
$$B_m = B_{m-1} \vee A_m = A_1 \vee A_2 \vee \ldots\ldots\ldots \vee A_m$$

Theorem 1

The i-j entry of A_m is 1 if it is possible to proceed from box i to box j in exactly m steps, and 0 otherwise. The i-j entry of B_m is 1 if it is possible to proceed from box i to box j in at most m steps, and 0 otherwise.

Proof: For $m = 1$, both statements reduce to definitions. Now suppose both statements hold for $m = r$; and consider the case $m = r + 1$. The i-j entry of A_{r+1} is just $\bigvee_k (c_{ik} \wedge a_{kj})$ where c_{ik} denotes the i-k entry of A_r. This is *zero*, unless for some k we have $c_{ik} = a_{kj} = 1$. But this means that it is possible to proceed from box i to box k in exactly r steps, and from box k to box j in exactly one step. Thus the i-j entry of A_{r+1} is 0 unless it is possible to proceed from box i to box j in exactly $r + 1$ steps. The second statement follows immediately from the first.

Theorem 2

The limit $\lim B_m$ as $m \to \infty$ exists as a Boolean matrix, which we denote by B. Moreover, we have $B = B_m$ for all $m \geq p$, where p is the length of the longest open path in the diagram.

Proof: Since the entries of B_m are monotone increasing with m, it is clear that $\lim B_m$ as $m \to \infty$ exists and forms a Boolean matrix. The second statement follows from the observation that if it is possible to proceed from box i to box j at all, it is possible to do so along an open path (*i.e.*, one containing no loops), and hence in less than $p + 1$ steps. Thus if the i-j entry of B_m is 1 for *any* m, it is 1 for $m = p$. This means that $B_m = B_p$ whenever $m \geq p$.

Theorem 3

The i-j entry of B is 1 if it is possible to proceed from box i to box j in any number of steps, and 0 otherwise.

Proof: This follows immediately from the proof of Theorem 2.

The matrix B is obviously computable by machine from the matrix A, and since only Boolean operations are involved, the time required for this computation is not prohibitive even for fairly large n. On the other hand, it follows from Theorem 3 that the matrix B contains detailed information about the consistency of the flow diagram. We cite some obvious examples:

(1) It is possible to get from the input to box i only if $b_{0i} = 1$. Thus if there are no spurious boxes, the top row of B must contain all 1's (except for b_{00}).

(2) It is possible to get from box i to the output only if $b_{i(n+1)} = 1$. Thus if there are no boxes without exits, the last column of B must contain all 1's (except for $b_{(n+1)\,(n+1)}$).

(3) It is possible to get from box i to box i only if $b_{ii} = 1$. Thus if there are no loops in the program, the main diagonal of B must contain all 0's. Boxes involved in loops are represented by 1's on this diagonal.

(4) After leaving box i, it is possible to go through box j only if $b_{ij} = 1$. Now if we alter box i then only those boxes following box i in the program will be affected. These boxes are represented by 1's in the ith *row* of B.

(5) If the matrix decomposes into relatively independent submatrices, then the program decomposes into relatively independent subprograms. Thus it may be possible to identify natural subprograms directly from the form of the matrix B.

Examples

The foregoing theory will be further illuminated by application to concrete problems. As a first example we choose a flow diagram containing an obvious inconsistency, and show how this inconsistency is

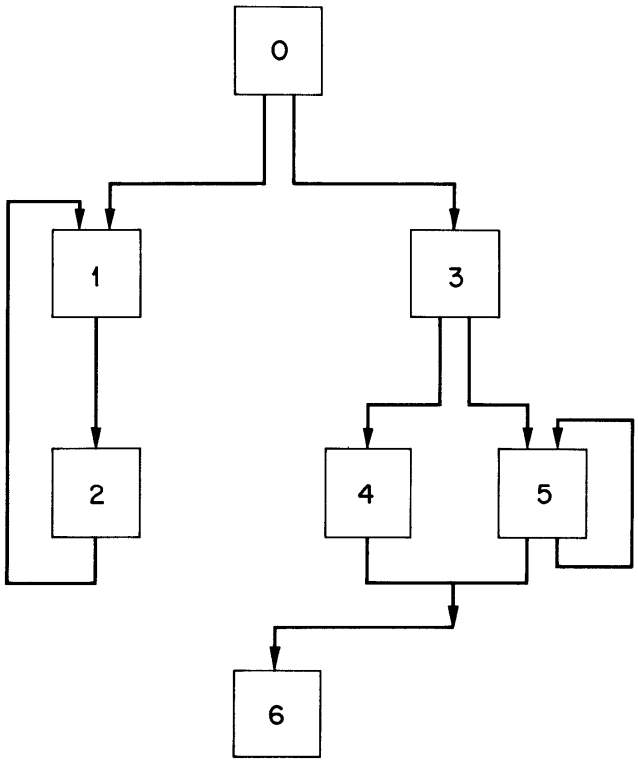

Fig. 1.

reflected in the matrix B. The diagram is shown in Fig. 1. Here the boxes are already numbered, including the input and output boxes. The connectivity matrix for this diagram is a 7×7 matrix, whose entries are

$$A = \begin{matrix} 010\ 100\ 0 \\ 001\ 000\ 0 \\ 010\ 000\ 0 \\ 000\ 011\ 0 \\ 000\ 000\ 1 \\ 000\ 001\ 1 \\ 000\ 000\ 0 \end{matrix}$$

Now $A_1 = B_1 = A$. Straightforward computation gives

$$A_2 = A \wedge A = \begin{matrix} 001\ 011\ 0 \\ 010\ 000\ 0 \\ 001\ 000\ 0 \\ 000\ 001\ 1 \\ 000\ 000\ 0 \\ 000\ 001\ 0 \\ 000\ 000\ 0 \end{matrix}$$

$$B_2 = B_1 \vee A_2 = \begin{matrix} 011\ 111\ 0 \\ 011\ 000\ 0 \\ 011\ 000\ 0 \\ 000\ 011\ 1 \\ 000\ 000\ 1 \\ 000\ 001\ 1 \\ 000\ 000\ 0 \end{matrix}$$

$$A_3 = A_2 \wedge A = \begin{matrix} 010\ 001\ 1 \\ 001\ 000\ 0 \\ 010\ 000\ 0 \\ 000\ 001\ 0 \\ 000\ 000\ 0 \\ 000\ 001\ 0 \\ 000\ 000\ 0 \end{matrix}$$

$$B_3 = B_2 \vee A_3 = \begin{matrix} 011\ 111\ 1 \\ 011\ 000\ 0 \\ 011\ 000\ 0 \\ 000\ 011\ 1 \\ 000\ 000\ 1 \\ 000\ 001\ 1 \\ 000\ 000\ 0 \end{matrix}$$

A glance at the diagram shows that all possible paths (without repetition) can be traversed in at most three steps, so that by Theorem 2, $B = B_3$. This can be checked by computing B_4, which is equal to B_3. From this matrix we verify immediately that all boxes are connected to the input (first row), but boxes 1 and 2 are not connected to the output (last column). Boxes 1, 2, and 5 are involved in loops (main diagonal). Moreover, if we delete the first row and last column of B, then the remainder can be decomposed into submatrices:

$$\begin{matrix} 11\ 000 \\ 11\ 000 \\ 00\ 011 \\ 00\ 000 \\ 00\ 001 \end{matrix} \quad \begin{matrix} \\ M\ 0 \\ \\ 0\ N \\ \\ \end{matrix}$$

where $M = \begin{matrix} 11 \\ 11 \end{matrix}$ and $N = \begin{matrix} 011 \\ 000 \\ 001 \end{matrix}$ This implies that boxes 1 and 2 and boxes 3, 4 and 5 form two independent subprograms whose associated matrices are just M and N. (Of course, the simplicity of this decomposition is due to the particular scheme adopted for numbering the boxes.) This simple example serves to illustrate the scope of the method.

This same method has an obvious application to the problem of debugging programs already compiled. In this case the boxes are already numbered by the sequential description of the program. Moreover, it is not necessary to draw the corresponding flow diagram, since, except for transfers, each operation is followed by the next in sequence. As a second example we take a typical SAP writeup of an IBM 704 program, with no inconsistencies. (This program computes an array of 100 quantities c_{ij} according to the formula

$$c_{ij} = \begin{cases} A_i - B_j \text{ if } i > j \\ A_i + B_j \text{ if } i \leq j \end{cases}$$

SAP Program

```
 1. LXD   8
 2. SXD   4
 3. CLA  B1
 4. TXL   6
 5. CHS
 6. ADD  A1
 7. STO  C1
 8. TXI   9
 9. TXI  10
10. TNX   2
11. TXI  12
12. TNX   2
13. END
```

The associated connectivity matrix can be written down directly, and is simply

$$A = \begin{matrix} 010\ 000\ 000\ 000\ 0 \\ 001\ 000\ 000\ 000\ 0 \\ 000\ 100\ 000\ 000\ 0 \\ 000\ 010\ 000\ 000\ 0 \\ 000\ 001\ 000\ 000\ 0 \\ 000\ 000\ 100\ 000\ 0 \\ 000\ 000\ 010\ 000\ 0 \\ 000\ 000\ 001\ 000\ 0 \\ 000\ 000\ 000\ 100\ 0 \\ 010\ 000\ 000\ 010\ 0 \\ 000\ 000\ 000\ 001\ 0 \\ 010\ 000\ 000\ 000\ 1 \\ 000\ 000\ 000\ 000\ 0 \end{matrix}$$

(Note that, except for transfer instructions, 1's appear only on the super diagonal.)

The Precedence Matrix

A further analysis of the structure of a program can be made if information concerning the precedence relations in the program is available. If we know, for example, that the output of box i is required for the input of box j, then we know that the operation represented by box i must precede that represented by box j in the program sequence. Clearly this places additional requirements on the internal connectivity of the program.

The precedence relations may be incorporated into our analysis through the introduction of a second Boolean matrix C associated with the program, which we call the *precedence matrix*, (cf. [1, 9]). It is constructed as follows. We number the boxes of the diagram as in Section II, and stipulate that *the i-j entry c_{ij} of C is to be 1 if the output of box i (or any part of it) is required for the input of box j, and 0 otherwise.* Clearly this matrix contains the precedence relations in the same way that the matrix A contains the connectivity relations of the program, and will yield to a similar analysis. We observe here that the two matrices are closely related, though they need not be identical.

Proceeding as before, we define

$$C_m = C_{m-1} \wedge C$$
$$D_m = D_{m-1} \vee C_m$$
$$D = \lim_{m \to \infty} D_m$$

and observe that the results of that section may be translated immediately into the present situation. In particular, the i-j entry of the matrix D is 1 if and only if there is a chain of boxes in the diagram beginning with box i and ending with box j such that each box in the chain must precede the next. Obvious applications include the following:

(1) The precedence requirements are internally consistent only if the diagram contains no closed chain of boxes each of which must precede the next. This is the case only if no diagonal entry of D is 1. Thus we require that trace $D = 0$ for this consistency (cf. [1]).

(2) In general, box j depends on box i only if $d_{ij} = 1$. Thus if box i is altered, this will affect only those boxes whose entries in the ith row of D are 1.

(3) Occasionally it is desirable to reorder the sequence of operations in some part of the program. This is possible only if the precedence requirements are not violated by the reordering. Thus box i may be interchanged with box j in a chain of operations only if $d_{ij} = d_{ji} = 0$. Information of this kind is evidently useful in optimizing flow diagrams for time or storage requirements.

The Dominance Matrix

In studying problems involving the reordering of operations in a program, it is often useful to introduce a notion of *dominance* in the flow diagram, defined as follows: *We say box i dominates box j if every path (leading from input to output through the diagram) which passes through box j must also pass through box i.* Thus box i dominates box j if box j is subordinate to box i in the program. It may happen that two boxes dominate each other (in which case we say they are equivalent), or that neither dominates the other (in which case we say they are independent). The idea here, of course, is that reordering is possible only among boxes which are equivalent in this sense. Proceeding along these lines, we define a third Boolean matrix E, called the *dominance matrix*, by stipulating that *the i-j entry e_{ij} of E is 1 if box i dominates box i, and 0 otherwise.* It is clear that the dominance matrix is determined by the connectivity matrix, and can be produced from it by a suitable scanning procedure. Applications include:

(1) Box i and box j may be interchanged, precedence requirements permitting, only if they are equivalent. This is the case only if we have $e_{ij} = e_{ji} = 1$.

(2) In preparing a program for a machine which admits *parallel* operation, it is desirable to know which operations in the program may be performed simultaneously. Two operations may be performed simultaneously without further investigation only if they are equivalent and subject to no precedence requirements, *i.e.*, only if $d_{ij} = d_{ji} = 0$ and $e_{ij} = e_{ji} = 1$.

(3) It is sometimes useful to know when two programs are equivalent in some sense. Any effective definition of equivalence requires a detailed knowledge of what happens at branch points in the program (*i.e.*, the transfer conditions). An interesting analysis of this problem is summarized in [14], but does not seem readily adaptable to machine handling. By requiring a less effective definition of equivalence, we can give here an effective criterion for determining whether or not two programs are equivalent.

To be precise, let us agree that *two programs, containing the same operations subject to the same precedence requirements, are equivalent, if, for each path (leading from input to output) through the first, there is a corresponding path through the second passing through the same operations.* We do not require that the operations appear in the same sequence, or even that they appear the same number of times, in both paths. This definition, however, is sufficient for most purposes, at least for programs containing no loops; loops cannot be incorporated under

so simple a scheme, and require special consideration.

In terms of flow diagrams, the equivalence criterion may be stated as follows. *Two diagrams, made up of the same boxes subject to the same precedence requirements, are equivalent only if their dominance matrices are identical.*

Remarks

The essential point of our discussion is that the entire analysis given here can be readily performed on any (large-scale) digital computer. The feasability of computing the derived matrices B, D, and E by machine is assured for programs which are not too large. A very crude estimate indicates that the time required to compute B from A on the IBM 704 is of the order of 10 n^3 cycles, where n is the number of boxes in the diagram. In practice, this time may be reduced considerably by combining into one box any subroutine whose behavior is known. Thus for example it is advantageous to replace any chain of boxes by a single box. Similarly, in analyzing program writeups it is sufficient to consider only transfer operations. For instance, a reduced form of the matrix A of our second example is:

$$A' = \begin{matrix} 010\ 000\ 0 \\ 001\ 000\ 0 \\ 010\ 100\ 0 \\ 000\ 010\ 0 \\ 010\ 001\ 0 \\ 000\ 000\ 0 \end{matrix}$$

where boxes 1 through 9 have been combined in a single box.

Finally we remark that it is a straightforward problem to construct a debugging routine which could be used to analyze any program writeup whose transfer instructions have constant addresses. Such a routine would scan the writeup, enumerate the transfer instructions, construct the connectivity and dominance matrices from them, compute the derived matrices and point out any errors detectable by these methods. Thus the whole analysis becomes completely automatic.

Various other applications of this analysis are suggested by the results. By utilizing the evident adaptability of these matrices to computer handling, it is possible to construct automatic program analysis schemes which would detect in proposed programs a large class of common errors, isolate and identify key subroutines and reorganize them in optimal equivalent programs. Such a scheme is currently under investigation here at Lincoln Laboratory, MIT.

Bibliography

[1] E. W. Barankin, "Precedence Matrices," *Univ. of Chicago Management Sciences Research Project*, Research Report no. 26; December, 1953.

[2] I. M. Copi, "Matrix development of the calculus of relations", *Jour. Symbolic Logic*, vol. 13, pp. 193–203; 1958.

[3] W. Feller, "An Introduction to Probability Theory and its Applications," John Wiley and Sons, New York, N. Y., p. 350; 1957.

[4] F. Hohn and L. Schissler, "Boolean matrices and the design of combinational relay switching circuits," *Bell System Tech. Jour.*, vol. 34, pp. 177–202; 1955.

[5] M. Kac and J. C. Ward, "A combinatorial solution of the 2-dimensional Ising model," *Phys. Rev.*, vol. 88, pp. 1332–1337; 1952.

[6] G. Kron, "Tensor Analysis of Networks," John Wiley and Sons, Inc., New York, N. Y.; 1939.

[7] S. Lefschetz, "Topology," *Colloq. Publications Amer. Math. Society*, New York, N. Y.; 1930.

[8] R. D. Luce and A. D. Perry, "A Method of matrix analysis of group structures," *Psychometrika*, vol. 14, pp. 95–116; 169–190; 1949.

[9] R. B. Marimont, "A new method of checking the consistency of precedence matrices," *Jour. Assoc. Comp. Mach.*, vol. 6, pp. 164–171; April, 1959.

[10] J. Riordan, "An Introduction to Combinatorial Analysis," John Wiley and Sons, Inc., New York, N.Y.; 1958.

[11] D. Rosenblatt, "On the graph and asymptotic forms of finite Boolean relation matrices and stochastic matrices," *Naval Res. Logist. Quart.*, vol. 4, pp. 151–167; 1957.

[12] H. Seifert and W. Threlfall, "Lehrbuch der Topologie," Chelsea, New York, N. Y.; 1947.

[13] C. Shannon and W. Weaver, "The Mathematical Theory of Communication," Univ. of Illinois, Urbana, Ill.; 1949.

[14] Y. I. Yanov, "On matrix schemes," *Dokl. Akad. Nauk. USSR*, vol. 113, pp. 39–42; 1957.

Discussion

E. Fredkin (Bolt, Beranek and Newman): In the case of a closed subroutine used by more than one calling sequence, how do you represent the fact that, while many routines enter and many exit, the subroutine box may return only to the calling routine?

Mr. Prosser: Problems of this kind, of course, are not handled at all by this formalism. Nothing has been said about how you make the decisions about where to go. I have deliberately avoided this. Thus, the whole theory is a black box theory. Now actually in some flow diagrams there are paths which you cannot follow at all, because the appropriate combination of logical requirements is never satisfied. This formalism will not tell you that. The best it can do is tell whether there is a path going from here to there, without telling whether or not the conditions for it are met.

Mr. Shapiro (Nat'l Institute Health): Given that your formalism does not account for the nature of decision-making elements, what is the definition of equivalence?

Mr. Prosser: Well, there is an interesting problem here. Let me say, first of all, there is a study by the Russian mathematician, Yanov, who has made a definition of equivalence which says, roughly speaking, that two programs are equivalent if they go through the same boxes in the same order. That is, for each path through one program, there is a path through the other one which does the same operations in the same order. In order to prove statements like that, you have to know something about how many times you go around loops, and I have no way of counting this in the present formalism.

So the definition of equivalence which I'm using here must be very weak. It would run something like this, and this is, in fact, the precise statement to which I was referring: two diagrams are equivalent if, for every path through the first one, there is a path through the second one which goes through the same boxes, not necessarily

in the same order and not necessarily the same number of times. But they do the same things. You recognize that this definition is quite weak unless you know something more about the number of times you go around loops. Using this for a definition, then the statement is that two diagrams are equivalent only if their dominance matrices are identical.

Mr. Miller (MITRE): Do you have a way of automatically generating your first matrix?

Mr. Prosser: You can do this in some cases, but not from a diagram. You can do it, for example, from an SAP write-up or from certain other kinds of write-ups automatically, providing that certain restrictions are placed on the write-ups — that they not be too complicated. As far as flow diagrams go, there is an obvious problem here. If you work from a flow diagram, you have to try somehow or other to get the diagram into the machine. We don't really know how to do this effectively, but then we really haven't studied the question. What we've done in actually running this experimentally is to take a typical flow diagram and try to draw these matrices by hand. All you have to do is record the 1's, of course. I don't know how to do it automatically.

I would like to say, though, that this program which I refer to, which computes the derived matrix B, we intend to submit to SHARE, so it ought to be available to the computing world fairly soon.

C. E. Dorrell (IBM): Do you have a program to compute F, the dominance matrix, and, if so, what is its running time?

Mr. Prosser: This is in the process of being put together. It should take about the same running time.

H. D. Friedman (Technical Operations): Since B is a geometric series of powers of A, although in the Boolean sense, isn't there an analytic method for obtaining B?

Mr. Posser: The question can be rephrased this way. Let B_K be the Kth step of the B matrix. How far out do you have to go before you have reached the limit? As I have indicated already, there is a number such that, beyond that, the B_K's are already constant and are equal to the limiting matrix. How far out is it? Well, an upper bound is the length of the longest path through the diagram, which is always less than the number of boxes in the diagram. Actually, our routine doesn't compute B by the process which I showed on the slide. If you look at the matrix A plus A square, and raise this to high powers, it turns out that this process gives you B. Now for a 500 by 500 matrix, something like 2^9 powers is enough, so the maximum number of squarings required is something like nine. So there is an upper bound which is not too big. The thing which makes this feasible, of course, is that the matrices are all zero's and one's. It is much easier to do matrix operations with them than with the usual matrices with arbitrary entries.