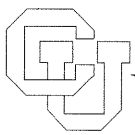
An Algorithm for Finding the Elementary Circuits of a Directed Graph *

Andrzej Ehrenfeucht
Lloyd D. Fosdick
Leon J. Osterweil

CU-CS-024-73



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Supported in part by National Science Foundation Grant #GJ-36461.

An Algorithm for Finding the Elementary Circuits of a Directed Graph*

by

Andrzej Ehrenfeucht
Lloyd D. Fosdick
Leon J. Osterweil

Report #CU-CS-024-73 August 1973

* Supported in part by National Science Foundation Grant # GJ-36461

1. INTRODUCTION

This paper presents an algorithm for finding all the elementary circuits in a directed graph. The method employed is a hybrid depth-first and breadth-first search. The correctness of the algorithm is proven by means of a novel informal proof technique, and the efficiency of the algorithm is studied carefully.

Tiernan [1], Weinblatt [2], and Tarjan [3] have recently produced algorithms for finding all the elementary circuits in a directed graph. Except for Tarjan, these investigators have not given precise efficiency measures of their algorithms. In Tarjan's paper crippling weaknesses are demonstrated in the algorithms of Tiernan and Weinblatt. He also asserts [†] that the efficiency of his algorithm is $O(n \times e)$. In this paper, we too establish $O(n \times e)$ as the measure of our algorithm's efficiency and also develop some heuristic machinery for speeding the execution of the algorithm even further. This algorithm differs significantly from Tarjan's and is noteworthy for its straightforward simplicity.

The major problems to be solved in producing an efficient algorithm are: 1) to find all elementary circuits, 2) to avoid finding any circuit more than once, and 3) to avoid wasting a great deal of effort in a fruitless search for nonexistent circuits. The depth-first search technique proves to be an excellent vehicle for solving problems 1) and 2). The depth-first search is a technique which has been employed in the past in such places as garbage collection marking algorithms, and has more recently been employed with great success by Hopcroft and Tarjan [4] in the production of highly efficient graph algorithms. An essential feature of all depth-first graph searches is

---

† Private communication. The efficiency measure presented in reference 3 is incorrect. Here n and e are the number of vertices and edges, respectively.

a path stack which, as the search proceeds, will always contain a simple path through the graph. A new point will be added to the stack provided that it can lengthen the existing simple path, but a point will be popped from the stack only when all simple paths which lead from it and do not intersect the simple path leading to it have been created on the stack.

It is rather easy to use this method to produce all elementary circuits, for any unsuccessful attempt to extend the existing simple path on the path stack must necessarily produce an elementary circuit. Hence, the depth-first search, in its attempts to produce all simple paths, will also generate all elementary circuits. Moreover, through careful and systematic selection and screening of points to be considered for placement upon the pathstack, it is fairly easy to guarantee that no elementary circuit is generated more than once.

Thus it is not surprising that other workers (e.g., Tarjan, and Tiernan) have used modified depth-first search methods to find all cycles. All have attempted to optimize their searches through the elimination of fruitless searching. They have attempted to do so without departing from the depth-first search philosophy. We have also begun with a depth-first search but have attacked the third major problem by departing from the basic depth-first approach and using a breadth-first approach.

The essential difference here is that this algorithm recognizes that the goal of the depth-first search should be the production of elementary circuits, not simple paths. Thus whenever the algorithm is about to add a vertex to its path stack it first determines whether the addition of the candidate vertex will further the production of a circuit. In particular, it is necessary to determine whether there exists a simple path from the candidate vertex to

the vertex on the bottom of the path stack which does not intersect the path stack. A breadth-first search is well suited to efficiently solving the more general problem of identifying all vertices reachable from a candidate vertex by paths not intersecting the path stack. Hence it is used here.

The search begins from an initial vertex and it is assumed that every other vertex on the graph lies on some path from this vertex. If this assumption were false, then the algorithm would apply on the maximal subgraph for which it were true; in this case it would be a simple matter to use the algorithm iteratively until the entire graph was considered.

As already noted, the efficiency estimate for this algorithm equals the estimate for the best known existing algorithm (Tarjan's). Tiernan's work has, however, helped convince us of the worth of heuristics. Thus, in a preliminary pass, this algorithm gathers some heuristic information which in many cases speeds the execution of the algorithm considerably. This information is in the form of a triangular Boolean array. By consulting this array, the algorithm can often immediately recognize that certain vertices cannot possibly lead to the production of new simple cycles. Thus the array can often preclude the necessity for breadth-first searches.

2. SOME DEFINITIONS FROM GRAPH THEORY

Before proceeding any farther, it is important that we define the graph theoretic structures with which we will be working.

Definition 2.1: A vertex set, V, is a set of vertices, or points.

Definition 2.2: The edge set on V, E_V is the set of all ordered pairs of vertices in V, specifically including the vertex pairs for which both vertices are the same.

Definition 2.3: A graph is an ordered pair (V, E_V) where V is a vertex set and E_V is some subset of E_V, the edge set on V.

Definition 2.4: The undirected edge set on V, F_V, is the set of all unordered pairs of distinct vertices in V.

Definition 2.5: An undirected, loop free graph is an ordered pair (V, F_V) where V is a vertex set and F_V is some subset of F_V, the undirected edge set on V.

It is important to note here that our definitions are somewhat non-standard. Most authors define a graph to be what we have defined as an undirected, loop-free graph. Moreover, what we define to be a graph is more commonly referred to in the literature as a directed graph with loops. Since our paper deals almost exclusively with the latter structure, we have formed our definitions as we did largely as a matter of convenience.

Definition 2.6: Let $G = (V, E_V)$ be a graph. Then if $e \in E_V$, we call e an edge of the graph G, moreover, if we let e = (a, b), then a, b are vertices of G, and we refer to a as the source vertex or tail of e and b as the destination vertex or head of e. The total number of edges having the tail vertex a is called the out-degree of a. The total number of edges having the head vertex b is called the in-degree of b.

Definition 2.7: Let $G = (V, E_V)$ be a graph. Suppose $a, b \in V$. We define a path in G from a to b to be a sequence of vertices $a = v_0, v_1, v_2, \ldots, v_{n-1}, v_n = b$ for which v_i V, $i = 0, 1, \ldots, n$ and for which $(v_{j-1}, v_j) \in E_V$, $j = 1, 2, \ldots, n$. An elementary path in b from a to b is a path in G from a to b for which all the v_i are distinct except perhaps $v_0 = v_n$.

Definition 2.8: Let $G = (V, E_V)$ be a graph. A circuit in G is a path in G from a to a, $a \in V$.

Definition 2.9: Let $G = (V, E_V)$ be a graph. An elementary circuit in G is an elementary path in G from a to a, $a \in V$.

3. THE ALGORITHM

An informal description of the algorithm is presented here, along with an example showing how execution proceeds for a sample graph. Flow diagrams are presented in the following section where they are used in connection with proving certain assertions about the algorithm.

The algorithm is divided into two phases: The first phase gathers information about the graph and the second phase actually finds the circuits of the graph with the help of this information. Specifically, in the first phase, a depth-first search is used to investigate the structure of the graph. In the course of the search, the vertices of the graph are numbered using an order which facilitates the functioning of the algorithm, and a lower triangular Boolean array, E, is produced. We use the vertex numbering to classify the circuits of the graph according to their lowest numbered vertex. Hence the problem of phase 2 is reduced from the problem of finding all circuits of a graph to the problem of finding all circuits of a given class. This reduction makes it easier to avoid finding a given circuit more than once during phase 2, a potential source of inefficiency.

The purpose of the E array is likewise to help avoid inefficiency in phase 2. Phase 1 sets an entry, E[i, j] of E, to be TRUE if and only if there is a path from the vertex numbered i to the vertex numbered j, where j is the lowest numbered vertex of some circuit. All other entries in E are set FALSE. Using the E array as a guide, we can avoid examination of certain irrelevant parts of the graph during phase 2.

The second phase is essentially a depth-first search whose efficiency is greatly improved by the superposition of a breadth-first search. In this phase

we find all circuits of class J setting J successively to 1, 2, ..., n where n is the number of vertices of the graph. The procedure for finding all circuits of class J works roughly as follows: A path from vertex J is developed using a depth-first search. The path is extended to a vertex i only if the resulting path is part of a new circuit. A set of tests is applied to determine whether vertex i can be added to the path. Some of these tests are independent of the number of vertices or edges in the graph and they are applied first. If all of these tests are passed then a breadth-first search from i is initiated to determine whether there is a path from i back to J which does not intersect the path already developed. If this test is passed then the path is extended to vertex i and the depth-first search continues from vertex i. Whenever the depth-first search encounters vertex J the path which has been developed is a circuit and is reported.

The orderly functioning of both phases of the algorithm relies on a systematic procedure for searching along the edges from a given vertex. Hence, in order to understand the detailed description of the algorithm which follows, it is important to understand the mechanism which is used to select edges from a given vertex. Let us assume v is a vertex with out-degree k_v. We call e_v, an integer on the set $\{0, 1, \ldots, k_v\}$, the edge selector from v; each of its values 1, 2, ..., k_v represents an edge with tail v. When $e_v = 0$ we say that the edge selector is in the null state. Both phase 1 and phase 2 begin with all edge selectors in the null state. Selection of a new edge from v consists in incrementing e_v, modulo $k_v + 1$; thus if $e_v < k_v$ then $e_v \leftarrow e_v + 1$ otherwise $e_v \leftarrow 0$. The null state of the edge selector is used by the algorithm as a signal to initiate backtracking.

The first phase of the algorithm functions in the following way. A vertex is selected, it is numbered 1, and it is put on an initially empty stack, P; this action and the subsequent steps are displayed in a series of pictures in Figure 3-1. The selection of this initial vertex is not entirely arbitrary, for it is assumed that every other vertex on the graph lies on some path from the initial vertex. The vertex at the top of the stack is called POINT: it is indicated by the symbol ■ in Figure 3-1. An edge from POINT is selected as described above and the head of the edge is called NUPT; NUPT is indicated by the symbol ● in Figure 3-1. We assume, for the time being, that selection of the edge does not put the edge selector in the null state. Exactly one of the following three conditions will thus be true:

(a) NUPT is not numbered;

(b) NUPT is numbered and is on P;

(c) NUPT is numbered and is not on P.

If (a) is true, then NUPT is numbered, it is pushed on the stack, it becomes POINT cf. Figure 3-1-3, and we continue by selecting an edge from POINT as above, cf. Figure 3-1-4. If (b) is true, then a circuit has been traversed and, if J is the number of NUPT, the circuit belongs to class J because the vertex numbering system guarantees that every vertex on the circuit other than NUPT is numbered higher than J; cf. Figure 3-1-8. The circuit is not reported, instead the element E[i, J], where i is the number of POINT, is marked TRUE; cf. Figure 3-1-8 where x stands for TRUE. The process continues with the selection of the next edge from POINT. Finally, if condition (c) is TRUE the ith row of E, E[i,•], is changed by the assignment

$$E[i, k] \leftarrow E[i, k] \vee E[r, k], \quad (k = 1, 2, \ldots, r) \tag{3-1}$$

where r is the number of NUPT.

In this way circuit information associated with NUPT is inherited by POINT, cf. Figure 3-1-14. Once again the process continues by selecting the next edge from POINT.

Concluding the discussion of the first phase, let us consider the event in which selection of a new edge from v puts the edge selector e_v in the null state. This event indicates that every edge from v has been selected once since e_v was last in the null state. When this occurs backtracking begins. In the backtracking procedure the stack P is popped so that the predecessor of POINT now becomes POINT and circuit information is inherited by the new POINT from the old POINT as shown in Equation (3-1); cf. Figure 3-1-9.

It is important to note that the old POINT's circuit information must, at this time, also be carried back to points other than the new POINT. Specifically, if t is a vertex which had previously been popped off the stack, P, and if the old POINT is reachable from t, then t must also inherit the circuit information of the old POINT. This is true because if the old POINT lies on a circuit of class J, then t might also lie on a circuit of class J (if $J \leq t$). Fortunately all such vertices, t, are readily identifiable. They are exactly those vertices, t, for which $E[t, v_0]$ (where v_0 is the number of the old POINT) is TRUE at the time when the old POINT is to be popped off of P. Thus at this time we carry back circuit information not only to the new POINT, but also to all vertices t, for which $E[t, v_0]$ is TRUE; specifically, for all $t > v_0$ such that $E[t, v_0]$ = TRUE perform the assignment

$$E[t, k] \leftarrow E[t, k] \vee E[v_0, k] \quad (k = 1, 2, \ldots v_0 - 1); \tag{3-2}$$

cf. Figure 3-1-12.

When the stack contains only vertex 1 and edge selection puts the edge pointer for this vertex in the null state the first phase of the algorithm terminates and the second phase begins, cf. Figure 3-1-15. It is worth noting that every edge selector will be in the null state at this time.

The second phase of the algorithm functions in the following way. A cycle class indicator, J, is initially set equal to 1 indicating the beginning of the search for all cycles of class 1; this action and the subsequent steps are displayed in a series of pictures in Figure 3-1. We observe that vertex 1 is on P, having been left there upon completion of the first phase, cf. Figure 3-1-16. Thus, vertex 1 is POINT.

The entry E[J, J] is examined. If E[J, J] is FALSE then there are no cycles in this class, and no further searching will take place. Let us assume for the time being, however, that E[J, J] is TRUE, implying that there is at least one cycle in class J. An edge from POINT is selected, the head being identified as NUPT with vertex number i, it being assumed now that the edge selection does not put the edge pointer in a null state. Exactly one of the following conditions will be true:

(a) $i = J$;

(b) $i < J$;

(c) $i > J$ and E[i, J] is FALSE;

(d) $i > J$ and E[i, J] is TRUE.

If condition (a) is true, then a cycle of class J has been found and it is reported at this time; cf. Figure 3-1-23. If (b) is true then NUPT cannot be on a cycle of class J and another edge is selected from POINT; cf. Figure 3-1-43. If (c) is true the situation is the same as when (b) is true.

If (d) is true NUPT may be on a cycle of class J now partially formed in P and a further condition is examined:

(e) NUPT is on P.

If (e) is true, then a cycle has been found but it does not belong to the class under consideration. It is ignored at this time and a new edge is selected. If (e) is false, we have not ruled out the possibility that NUPT lies on a new circuit of class J. Unfortunately, neither have we established that NUPT must necessarily lie on such a circuit. We know that there are one or more elementary paths from NUPT back to vertex J, but we do not know whether any of these paths is entirely free of points currently on P. If so, we should put NUPT on P make it POINT and continue the process described above, cf. Figure 3-1-18. If not, we should select a new NUPT and continue as above, cf. Figures 3-1-21 to 3-1-23. It is here that a breadth-first search technique is used to give the desired answer.

The breadth-first search uses a list P'. Whenever the state of the path stack P is changed, P' is cleared. The breadth-first search begins by placing a vertex v on P'; a pointer, I, is used to identify the head of the list P'. Another pointer I' is set to v's position on P'. The head, V' of each edge from P'[I'] is examined; if the number of V' is J then the search terminates successfully; if not and V' is not on P' or on P it is added to the head of the list P' (and I is incremented by one); if not and V' is on P or on P' then no further action on V' is taken. After all vertices adjacent from P'[I'] have thus been considered I' is incremented by one and the above search from P'[I'] is repeated. This continues until "success" or until I' > I, signifying failure. If this search ends successfully we know that

there is a simple path from the vertex at the bottom of P' to the vertex at the bottom of P which does not cross any vertex on P. The determination of whether a vertex is on P or P' can be done easily by addressing a Boolean array, W[m], where W[m] = T if m is on P or P', otherwise W[M] = F. It is worth noting at this point that if the search terminates unsuccessfully, we will soon be embarking upon another breadth-first search with the same J, the same POINT, but a new NUPT. The information currently in P' and W will be useful to us at that time. Thus we save it. By the construction of this algorithm, it is certain that we will eventually encounter a NUPT from which the search for J will succeed. After this has happened we can (indeed must) reinitialize P' and W.

By using this breadth-first search, we are assured that a vertex will be pushed on P during phase 2 if and only if it is a part of a new elementary circuit.

When selection of an edge from POINT puts the edge pointer in the null state and the number of the vertex POINT is unequal to J backtracking occurs; if POINT is equal to J, then J is incremented by 1 and if $J \leq n$ the above process is next applied to find all circuits in class J cf. Figure 3-1-35; if $J > n$ the second phase terminates. Backtracking in the second phase consists in popping the stack P, with the new occupant of the top of the stack becoming POINT, selection of the next edge from POINT, and continuation of the search as described above, cf. Figures 3-1-23 to 3-1-25. When the second phase terminates all circuits have been reported exactly once and execution of the algorithm terminates.

4. PROOF OF TERMINATION AND REPORTING ALL CIRCUITS

Description of the space-time notation. In this section a series of lemmas and theorems is proved leading to a proof of termination and a proof that all cycles are reported once. A discussion of efficiency is deferred until Section 5. Our method of proof makes reference to a flow diagram in which every box is uniquely numbered, and to a hypothetical clock which is initially zero and which advances one unit as control moves through a box. The flow diagram is constructed with only one line entering a box and one or more lines leaving a box. If more than one line leaves the box they are appropriately labeled for identification. The notation (b-) is used to denote a location on the line entering box b and (b+) to denote a location on the line leaving box b. If more than one line leaves a box b, say two lines labeled T and F, the notation (bT) and (bF) is used to designate locations on these lines. When reference to a location (b-) or (b+) at a particular time, say t, on the hypothetical clock is made, the notation (b-, t) or (b+, t) is used. Thus, we are using a space-time coordinate system to identify points of the computation.

These conventions are illustrated for the simple flow diagram in Figure 4-1. The location (2+) refers to a location on the line leaving box 2 and the following assertion is obviously true:

$$I = 1 \quad \text{at} \quad (1+)$$

or more simply

$$I(1+) = 1$$

When no reference to time is made, as in the above assertion, it is understood

to apply to whatever time (or times) at which the location is reached. The clock is zero at (0+) by our conventions so the following assertion is also true:

$$I(1+, 1) = 1$$

Note that the clock advances one unit in passing through box 1. The locations (1+) and (2-) are equivalent: assertions at one of these hold at the other. On the other hand, (2+) and (3-) are not equivalent locations. The assertion

$$X(2+) = 1$$

is true, but the assertion

$$X(3-) = 1$$

is false. However, the assertions

$$X(3-, 2) = 1, \quad X(3-) \geq 1$$

are true. The assertion

$$I(4F) \neq 10$$

is true, but the assertion

$$I(4F, 4) = 1$$

is false.

It is convenient to use the symbol τ to denote the interval of time between successive visitations of a location on the flow diagram. Thus, we make statements such as:

$$\text{If } I(3+, t) = 7 \text{ then } I(3+, t + \tau) = 8$$

In our usage τ is not a constant, it is simply a notational device to identify the time between successive visits to a location on the flow diagram.

Points will often be characterized as feasible or necessary. A feasible point is a point which may be attained in the course of execution of the

algorithm. To illustrate these two concepts, suppose the constant ten in box 4 of Figure 4-1 is replaced by the variable symbol N. We assume that when this algorithm is called into execution N will have been assigned a particular value, a positive integer. In considering the algorithm, with N unassigned, we would say that the points (4T, 4) and (4F, 4) are feasible since (4T, 4) is reached if N = 2 and (4F, 4) is reached if N $\geq$ 2. On the other hand (4T, 3) and (4F, 3) are not feasible since these points cannot be reached for any value of N. The point (2+, 2) is necessary since it is reached for all values of N. In this example, once a value for N is assigned the course of execution is completely determined and all points become either necessary or not feasible.

Description of the flow diagram for the first phase. A flow diagram for the first phase of the algorithm is shown in Figure 4-2. In the discussion which follows the expression vertex identifier is used to refer to the number initially used to identify the vertex in the graph before execution of the algorithm, and the expression vertex number is used to refer to the number assigned to the vertex by the algorithm. Vertex identifiers and vertex numbers are integers in the set {1, 2, ..., n}, where n is the number of vertices in the graph. The letters T and F are used throughout to denote truth values, TRUE and FALSE. The meaning of the variables appearing in the flow diagram follows:

E: lower triangular Boolean array, described in Section 3. E is indexed by vertex numbers.

K: path stack pointer; the value of K is the position of the top of the stack P.

J: vertex numbering parameter; sequential values (1, 2, ...) of J are used to number the vertices.

L: array of vertex numbers; L[1], L[2], ..., L[n] represent the n vertices of the graph; the value of L[i], where i is the vertex identifier, is the number assigned to vertex i; if L[i] = 0 then vertex i is not numbered.

P: path stack; P[K] is the top of the stack; the value of P[j], $1 \leq j \leq K$, is a vertex identifier; P[1], P[2], ..., P[K] is a simple path; P[K] is POINT (cf. section 3).

V: NUPT vertex (cf. section 3); the value of V is a vertex identifier; the value V = 0 occurs when the edge selector for POINT (i.e., P[K]) is in the null state.

I: the initial vertex; the value of I is the identifier of the initial vertex, specified on entry to the first phase.

The meaning of the subroutines appearing here follows:

SEL(v): select a new edge from v; the value of SEL(v) is the identifier of the head vertex of an edge with a tail vertex having identifier v (cf. section 3).

COPY (u, v): u and v are vertex identifiers, and execution of this subroutine performs the assignment shown in (3-1) with i = L[u], J = [v].

COPY* (u, v): perform the assignment shown in (3-1) with i = L[u] and J = L[v] and also the set of assignments shown in (3-2) with v_0 = L[v].

A brief description of each box in the flow diagram follows:

0: Execution begins at this point, I has been assigned a value which is the vertex identifier for the initial vertex in the graph.

1: Initialization takes place here. Set all entries in the E array FALSE ($E \leftarrow F$), mark all vertices as unnumbered ($L \leftarrow 0$), number the initial vertex ($J \leftarrow 1$, $L[I] \leftarrow J$), put the initial vertex on the path stack ($K \leftarrow 1$, $P[K] \leftarrow I$).

2: Select an edge from the vertex at the top of the path stack. Set V equal to zero if the edge-pointer is put in the null state, otherwise set V equal to the vertex identifier for the head of the selected edge.

3. Termination test. The algorithm terminates when there is only one vertex in the path stack ($K = 1$) and the edge-pointer for that vertex is in the null state ($V = 0$).

4: Check if the edge pointer for the vertex at the top of the path stack is in the null state, $V = 0$ is TRUE if it is.

5: Backtrack. Copy information in the E array according to (3-1 and 3-2) and pop the path stack ($K \leftarrow K-1$).

6: Check if the vertex V is on the path stack, $V \subset P$ is TRUE if it is.

7: An edge from P[K] has a head in the path stack, thus a cycle has been found; set an element in the E array equal to TRUE to "remember" this cycle.

8: Check if V is a numbered vertex; $L[V] \neq 0$ is TRUE if it is numbered.

9: Number V ($J \leftarrow J + 1$, $L[V] \leftarrow J$) and put V on the top of the path stack ($K \leftarrow K + 1$, $P[K] \leftarrow V$).

10: Copy information from row L[V] of the E array into row L[P[K]] of the E array using (3-1), thus propagating circuit information from a vertex (V) which was previously on the path stack.

11: terminate execution of the first phase.

It should be pointed out that the test in box 6 does not require a search of the path stack. For example, an ordered list of n elements can be kept, each corresponding to a vertex and an element can be marked or unmarked as the corresponding vertex is added to or removed from the path stack.

Theorems and proofs for the first phase. Two lemmas which will be helpful in proving Theorem 1 on termination of the first phase of the algorithm are introduced first.

Lemma 1. If for some t, (2+, t) is necessary then either the assertion

$$(4.1) \qquad (V(2+, t) = 0) \wedge (K(2+, t) = 1)$$

is true or there is a t', $4 \leq t' \leq 6$, such that (2+, t + t') is necessary. If the assertion (4.1) is true, then there is no $t' > 0$ for which (2+, t + t') is feasible.

Proof of Lemma 1. If assertion (4.1) is true, then it is evident that there is termination at time t + 1 and so (2+, t + t') is not feasible for $t' > 0$. If it is false, then (3F, t + 1) is necessary. Now by simple enumeration, there are exactly four execution paths from box 4 to box 2: $(4 \to 5 \to 2)$; $(4 \to 6 \to 7 \to 2)$; $(4 \to 6 \to 8 \to 10 \to 2)$; $(4 \to 6 \to 8 \to 9 \to 2)$. From a consideration of the elapsed time on each of these paths the truth of this Lemma is evident. ∎

Lemma 2. $K(2+, t) \geq 1$ for all feasible points (2+, t).

Proof of Lemma 2. Note first that for the "earliest" feasible (2+, t), (i.e., that (2+, t) for which t is the smallest of all t for which (2+, t) is feasible) namely (2+, 2),

$$K(2+, 2) = 1.$$

Other than in the initialization box 1, K is changed only in boxes 5 and 9, by decrementing one unit or incrementing one unit. Hence, if

$$K(2+, t) = k,$$

then

$$K(2+, t + \tau) = k + 1, k, \text{ or } k - 1.$$

We now carry out a proof by contradiction. Suppose for some $(2+, t')$ we have $K(2+, t') < 1$. It is evident from the above observations that for this to happen there must have been a t'', $t'' < t'$, such that

$$K(5-, t'') = 1. \tag{4.2}$$

This implies, working backwards through box 4,

$$K(3F, t'' - 1) = 1,$$

$$V(3F, t'' - 1) = 0,$$

and, similarly,

$$K(3-, t'' - 2) = 1,$$

$$V(3-, t'' - 2) = 0.$$

But the last two relations and box 3 clearly show that the point $(3T, t'' - 1)$ is necessary, thus (4.2) and our hypothesis $K(2+, t') < 1$ are contradicted. ∎

Lemma 3. There is only a finite number of t_i for which $(9-, t_i)$ is feasible, namely $(9-, t_1)$, $(9-, t_2)$, ... , $(9-, t_p)$, where $p \leq n$.

Proof of Lemma 3. We note first that an entry in the L array is set equal to zero only in box 1 and for all $t > 0$ the point $(1-, t)$ is not feasible. Thus if at some later time $L[V]$ is set unequal to zero for some V, it will remain unequal to zero for all future times. We note that for each feasible $(9-, t_i)$ we have

$$L[V]\ (9-,\ t_i) = 0, \text{ and}$$

$$L[V]\ (9+,\ t_i + 1) \neq 0.$$

(Note that J is initially set to 1 (box 1) and subsequently changed only in box 9; it cannot be zero.) Thus a zero entry in the L array is removed, and since there are only a finite number of entries in the L array (equal to the number of vertices in the graph) the truth of the Lemma is evident.∎

Theorem 1. The first phase of the algorithm terminates; i.e., for some t' the point (3T, t') is necessary.

Proof of Theorem 1. By Lemma 3 there is a t" such that for all $t > t"$ the point (9-, t) is not feasible. Assume now that (3T, t') is not feasible for all $t' > t"$, then by Lemma 1 there is an infinite sequence $(2+,\ t_i)$ $i = 1, 2, \ldots$ of necessary points where $t_i < t_{i+1}$ and $t" < t_1$. Since there is a finite number of edges from each vertex and a new edge is selected on each execution of box 2 we see that there exists a t_i, say $t_{i'}$, such that

$$V(2+,\ t_{i'}) = 0,$$

$$K(2+,\ t_{i'}) = k.$$

We observe that $k \geq 1$ by Lemma 2. If $k = 1$, then $(3T,\ t_{i'} + 1)$ is necessary and the proof is completed. Otherwise, $k > 1$, and we see that $(5+,\ t_{i'} + 3)$ is necessary. Hence

$$V(2+,\ t_{i'} + \tau) = v,$$

$$K(2+,\ t_{i'} + \tau) = k - 1.$$

We now recall that k can only be incremented in box 9, and that we are only considering t for which $t > t"$ -- that is, points for which (9-, t)

is not feasible. Thus k cannot increase in value. Thus it is evident by induction that eventually, for some $t_{i''}$,

$$V(2+, t_{i''}) = 0,$$

$$K(2+, t_{i''}) = 1,$$

and it follows that $(3T, t_{i''} + 1)$ is necessary. We have shown that $(3T, t')$ is necessary for some $t' > t''$ provided that box 11 is not executed at some t for which $t \leq t''$. If 11 is executed for some $t \leq t''$ our assertion is clearly true, hence Theorem 1 is proven.∎

Now some lemmas are introduced which lead to the second theorem, concerning the state of the Boolean array E upon termination of the first phase.

Lemma 4. If I is the initial vertex and if there is a path from I to v, say $I = v_1, v_2, \ldots, v_p = v$, then v will be numbered in the first phase.

Proof of Lemma 4. The proof is by contradiction. Suppose v is not numbered then it can be shown that v_{p-1} was never placed on the path stack and so it was never numbered. This leads to a contradiction. The argument proceeds as follows. If v_{p-1} had been placed on the path stack, say

$$P(9+, t') = \ldots v_{p-1},$$

then termination of the algorithm implies that for some $t'' > t'$

$$V(2+, t'') = v_p.$$

Since, by our hypothesis, v_p is unnumbered, it is easily verified that $(9+, t'' + 5)$ is a necessary point, hence $v_p = v$ would be numbered. This contradiction forces the conclusion that v_{p-1} must not ever be stacked and hence remains unnumbered. Applying this argument recursively, working backward along the path, we will come to the conclusion that I was never placed on the path, an obvious contradiction.∎

It is to be noted that since we assume there is a path from the initial vertex to every vertex on the graph, it follows from this Lemma that in the first phase every vertex gets numbered.

Lemma 5. If there is a path $v_1, v_2, \ldots, v_p$ and v_1 has a lower number than the other vertices,

$$L[v_1] < L[v_j], \qquad j = 2, 3, \ldots p, \tag{4.3}$$

then in the interval (t', t") defined by

$$P(9+, t') = \ldots v_1, \tag{4.4}$$

and

$$V(2+, t") = 0,$$

$$P(2+, t") = \ldots v_1,$$

each of these other vertices will be placed on P.

Proof of Lemma 5. By the hypothesis (4.3) it is evident that these other vertices cannot have been on P before t' for then they would have been assigned a number less than $L[v_1]$. The proof is now just like that used for Lemma 4. Suppose v_p was not on P in (t', t") then it will follow that v_{p-1} was also not on P in (t', t") and tracing backward along the path we conclude v_1 was not on P in (t', t") violating (4.4). ∎

Lemma 6. If, at any time and for any i, j

$$E[i, j] = T,$$

then this equation will be true for all later times.

Proof of Lemma 6. The truth of this assertion is immediately evident from the observation that an entry in the E array can be set equal to F only in box 1 and (1-, t) is not feasible for $t > 0$. Notice that only operations performed in boxes 5, 7, and 10 can change an entry in the E

array and the change can only be from F to T. ∎

Now the second theorem concerning the first phase of the algorithm is presented.

Theorem 2. If there is an elementary circuit $v_1, v_2, \ldots, v_p, v_1$ of class J, (i.e., $L[v_1] = J$; $J < L[v_i]$, $i = 2, 3, \ldots, p$) then upon termination of the first phase

$$E[L[v_i], J] = T, \qquad (i = 1, 2, \ldots p). \tag{4.5}$$

Proof of Theorem 2. We will use an induction argument. It is easy to show that the hypothesis implies (4.5) is true for i = p. The argument proceeds in the following way. From Lemma 5 there is a time t_p, $t' < t_p < t''$, using the notation of Lemma 5, when

$$V(2+, t_p) = v_1,$$

$$P(2+, t_p) = \ldots v_1 \ldots v_p,$$

and it is easily seen that

$$E[L[v_p], J]\ (7+, t_p + 4) = T.$$

Now, using Lemma 6, it is evident that (4.5) holds for i = p. Let us assume now that (4.5) holds for i = s + 1, we will show that this implies it holds for i = s. Define t_s and t'_s

$$V(2+, t_s) = v_{s+1},$$

$$P(2+, t_s) = \ldots v_1 \ldots v_s,$$

$$V(2+, t'_s) = 0,$$

$$P(2+, t'_s) = \ldots v_1 \ldots v_s.$$

Earlier Lemmas can be used to establish that $(2+, t_s)$ and $(2+, t'_s)$ are both necessary. At the point $(2+, t_s)$ there are exactly three possibilities:

(a) $v_{s+1} \notin P \wedge L[v_{s+1}] = 0$;

(b) $v_{s+1} \notin P \wedge L[v_{s+1}] \neq 0$;

(c) $v_{s+1} \subset P$.

We consider each of these in turn. If (a) is true then

$$P(2+, t_s + \tau) = \ldots v_1 \ldots v_s\ v_{s+1}.$$

Now either

(a') $$E[L[v_{s+1}], J]\ (2+, t'_{s+1}) = T,$$

or

(a'') $$E[L[v_{s+1}], J]\ (2+, t'_{s+1}) = F.$$

If (a') is true then it is evident from the COPY* function and the fact that $(5-, t'_{s+1} + 2)$ is necessary that

$$E[L[v_s], J]\ (2+, t'_{s+1} + \tau) = T.$$

If, on the other hand, (a'') is true, there must be an edge from v_{s+1} to another vertex, v_q, on P, $L[v_1] < L[v_q] < L[v_{s+1}]$; thus

(4.6) $$E[L[v_{s+1}], L[v_q]]\ (2+, t'_s) = T.$$

Otherwise, there would be no possibility for $E[L[v_{s+1}], J]$ to be set equal to T at any later time. Furthermore, either

$$E[L[v_q], J]\ (2+, t'_q) = T,$$

or v_q is similarly linked to $v_{q'}$ where $L[v_1] < L[v_{q'}]$, for the same reason. Thus, eventually, for some vertex $v_{q''}$, with $L[v_1] < L[v_{q''}]$, in this chain we must have

$$E[L[v_{q''}], J]\ (2+, t'_{q''}) = T,$$

and by virtue of this chain

$$E[L[v_{s+1}], J]\ (2+, t'_{q''} + \tau) = T.$$

Observing that

$$E[L[v_s], L[v_q]]\ (2+, t'_{s+1} + \tau) = T$$

by (4.6) and the COPY* function it also follows that

$$E[L[v_s], J]\ (2+, t'_{q''} + \tau) = T.$$

If (b) is true then (10-, t_s + 4) is necessary and the COPY function will be performed. The argument here is the same as for case (a).

If (c) is true then it is evident that (7-, t_s + 3) is necessary, hence

(4.7) $$E[L[v_s], L[v_{s+1}]] \ (2+, t_s + \tau) = T.$$

Now consider the time t'_{s+1} which in this case satisfies $t'_{s+1} > t_s + \tau$. Either

(c') $$E[L[v_{s+1}], J] \ (2+, t'_{s+1}) = T, \text{ or}$$

(c") $$E[L[v_{s+1}], J] \ (2+, t'_{s+1}) = F.$$

In case (c') the COPY* operation and the link (4.7) produces

$$E[L[v_s], J] \ (2+, t'_{s+1} + \tau) = T.$$

In case (c") we follow an argument like that used in case (a"). Thus for every case we must conclude that (4.5) holds for i = s. This completes the induction argument, the theorem follows at once. ∎

Two observations concerning this theorem are worth noting here. First, it is evident that if there are no circuits of class J then E[J, J] is FALSE upon termination. Second, and not so evident, is that if E[i, J] is TRUE upon termination then it is not necessarily true that the vertex numbered i is on a circuit of class J. This is illustrated in Figure 4-3. This result is not as bad as it might seem at first, for in the course of looking for circuits of class 2 in the graph of Figure 4-3 during the second phase we will never encounter the vertex whose number is 5.

Description of the flow diagram for the second phase. A flow diagram for the second phase is shown in Figure 4-4. Except for J and I the meaning of the variables appearing in the flow diagram is the same as described for the first phase. Parameters which are newly introduced here are described below.

L*: array of vertex identifiers; L*[1], L*[2],...,L*[n] represent the n vertices of the graph; the value of L*[i], where i is the vertex number, is the vertex identifier.

J: Circuit class identifier, while J = 1 circuits of class 1 are being located, while J = 2 circuits of class 2 are being located, etc.

I: Index for head of path list P' which is maintained by BRFSRCH.

P': List of vertices encountered in the execution of BRFSRCH.

Subroutines BRFSRCH and REPORT introduced here are briefly defined below.

BRFSRCH: The breadth-first search algorithm (Figure 4-5). If there is a simple path from V to L*[J] which does not intersect a vertex on P then BRFSRCH is assigned the value T otherwise F.

REPORT: A subroutine to print or otherwise record the elementary circuit P[1], P[2],...,P[K], P[1].

A brief description of each box in the flow diagram follows:

0: Execution begins at this point, the array L* has been loaded with the vertex identifiers, the L and E arrays are in the same state as when the first phase terminated.

1: Initialize the circuit class identifier (J ← 0) and the path stack pointer (K ← 1).

2: Increment the circuit class identifier to commence search for the next class of circuits.

3: End test. Stop if circuit class identifier exceeds the number of vertices, n, in the graph.

4: Execution of the second phase terminates here.

5: Test for existence of circuits of class J; E[J, J] = F if there are no circuits of class J.

6: Put identifier for vertex J on the path stack and initialize P' pointer.

7: Select an edge from the vertex at the top of the path stack and set V equal to zero if the edge pointer is put in the null state, otherwise set V equal to the vertex identifier for the head of the selected edge.

8: End test for search of circuits belonging to class J. $V=0 \wedge K=1$ is TRUE if all circuits of class J have been considered.

9: Test for all edges examined from vertex at head of path stack.

10: Remove vertex from head of path stack, and initialize P' pointer.

11: Test if selected vertex is the identifier for vertex number J.

12: The path stack contains an elementary circuit, report it.

13: If any one of these four conditions is false V cannot be on the elementary circuit now being constructed.

14: If the breadth-first search algorithm returns a false value, then V cannot be on the circuit now being constructed.

15: V is on a circuit consisting of P[1], P[2],...,P[K],V and other vertices yet to be determined so put V on the top of the path stack, and initialize P' pointer.

Theorems and proofs for the second phase. It is important for an understanding of the subsequent proofs that the TRUE exit from box 14 will be taken if and only if V lies on a circuit of class J, the initial segment of which is on P upon entry to box 14. Furthermore, if the FALSE exit is taken then any

That is, at all times every vertex on P is different from every other vertex on P.

This lemma and the following one are easily proved by considering the implications of boxes 13 and 15. We omit the proofs.

Lemma 11. For all feasible points (7+, t) and all $1 < I \leq K$

$$L[P[I]]\ (7+,\ t) > L[P[1]]\ (7+,\ t) = J.$$

That is, at all times the lowest numbered vertex on P is the vertex at P[1] and that vertex is numbered J.

Lemma 12. For some k, (k = 1, 2, ..., n) the point (7+, t) with

$$K(7+,\ t) = k,$$

$$V(7+,\ t) = 0,$$

is necessary.

Proof of Lemma 12. If the Lemma were false there would be an infinite sequence of necessary points (7+, t), (7+, $t + \tau$), ($7 + t + 2\tau$), ... and K is nondecreasing on these points since box 10 can never be entered. Now the observation that a new edge has been selected on each successive entry to box 7 and the fact that the number of edges is finite shows that such an infinite sequence is impossible. ∎

Lemma 13. If the point (7+, t') with

$$K(7+,\ t') = k'$$

$$P(7+,\ t') = v_1\ v_2\ \ldots\ v_{k'}$$

$$V(7+,\ t') = 0$$

is necessary, then for no $t'' > t'$ will it be true that

$$P(7+,\ t'') = P(7+,\ t'). \tag{4.8}$$

Proof of Lemma 13. By Lemma 12 we know that there is some point (7+, t').

vertices on P' at the time of the exit cannot lie on a circuit of class J, the initial segment of which is on P. These assertions are proved later in the part which deals with the BRFSRCH algorithm.

Now a series of lemmas are introduced to prove Theorem 3 which is concerned with termination of the second phase. Some of these are similar to lemmas introduced already in dealing with the first phase and in these cases proofs are omitted.

Lemma 7. If for some t, (7+, t) is necessary, then either (7+, t + t') is necessary for $4 \leq t' \leq 3n + 7$ or (3T, t + t') is necessary for $3 \leq t' < 3n$.

Lemma 8. $K(7+, t) \geq 1$ for all feasible points (7+, t).

Lemma 9. If for some t', (11T, t') is necessary then P[1] (11T, t') = L*[J].

This lemma asserts that when a circuit is reported the number of the vertex at the tail of the list (i.e., in P[1]) is the number of the class of this circuit.

Proof of Lemma 9. Excluding box 1 which is never entered for $t > 0$, P is only changed in boxes 6, 10, and 15. Lemma 8 precludes changing P[1] in boxes 10 or 15. Since J is changed only in box 2, the only points (x, t) where

$$P[1] (x, t) \neq L^*[J]$$

are in the set (2+, t), (3F, t + 1), (3T, t + 1), (5F, t + 2), (5T, t + 2), assuming (2+, t) is feasible. Obviously (11T, t') for any t' is not in this set. ∎

Lemma 10. For all feasible points (7+, t) and all $1 \leq I < I' \leq K$

$$P[I] (7+, t) \neq P[I'] (7+, t).$$

Suppose the Lemma is false. Pick t" to be the earliest time after t' at which (4.8) holds; then we must have

$$K(7+, t'' - \tau) = k' - 1,$$
$$P(7+, t'' - \tau) = v_1, v_2, \ldots, v_{k'-1},$$
$$V(7+, t'' - \tau) = v_{k'},$$

but the properties of the SEL function imply then that for some t''', $t' < t''' < t''$

$$K(7+, t''') = k' - 1,$$
$$P(7+, t''') = v_1, v_2, \ldots, v_{k'-1},$$
$$V(7+, t''') = 0.$$

Now by following an obvious induction we conclude that there must be a time t'^{v}, $t' < t'^{v} < t''$ such that

$$K(7+, t'^{v}) = 1,$$
$$P(7+, t'^{v}) = v_1$$
$$V(7+, t'^{v}) = 0.$$

Now either the algorithm terminates without returning to box 7 or a new vertex replaces v_1 and v_1 can never again occupy P[1] (cf. Lemma 9 and the proof of Lemma 9). In either case there is an obvious contradiction. ∎

We are now in a position to prove the following Theorem.

Theorem 3. The second phase of the algorithm terminates; i.e., for some t', the point (3T, t') is necessary.

Proof of Theorem 3. Consider the possible paths from (7+, t) to (7+, $t + \tau$). For each of these one of the following is true (cf. Lemma 7):

(a) $K(7+, t + \tau) = K(7+, t) + 1$;

(b) $K(7+, t + \tau) = K(7+, t)$;

(c) $K(7+, t + \tau) = K(7+, t) - 1$.

If (a) is true then by Lemma 13 the path on P can never have been on P at any earlier time. Because there are only a finite number of possible paths it is evident that (a) can be true for only a finite number of points (7+, t). If (b) is true then after some finite amount of time either (a) will be true or (c) will be true or the algorithm terminates; this is evident from the observation that over the sequence of times for which (b) remains true a new edge is selected each time. If (c) is true then it will be true at some later time or the algorithm terminates. This is evident from the observations about (a) and (b) already made. But (c) cannot be true for an infinite sequence of times without violating Lemma 8. Hence, the second phase must terminate. ∎

Theorem 4. Only elementary circuits are reported.

Proof of Theorem 4. For any feasible point (11T, t)

$$P(11T, t) = v_1, v_2, \ldots, v_k$$

is clearly a path. By Lemma 10 it is an elementary path. Since

$$L[V](11T, t) = J,$$

then from Lemma 9, $V = v_1$ and $v_1, v_2, \ldots, v_k, v_1$ is an elementary circuit. ∎

Theorem 5. No circuit is reported more than once.

Proof of Theorem 5. Suppose for some t_1

$$P(11T, t_1) = v_1, v_2, \ldots, v_k,$$

$$V(11T, t_1) = v_1.$$

Then, since the algorithm terminates, there is a t' such that

$$P(7+, t_1 + t') = v_1, v_2, \ldots, v_k,$$

$$V(7+, t_1 + t') = 0.$$

Suppose that there is a t_2, $t_2 > t_1 + t'$, such that (11T, t_2) is a necessary point for which:

$$P(11T, t_2) = v_1, v_2, \ldots, v_k,$$
$$V(11T, t_2) = v_1,$$

then

$$P(7+, t_2 + t') = v_1, v_2, \ldots, v_k,$$
$$V(7+, t_2 + t') = 0,$$

and Lemma 13 is contradicted. ∎

Theorem 6. Every circuit is reported.

Proof of Theorem 6. This is easily proved by contradiction. Suppose $v_1, v_2, \ldots, v_p, v_1$ is a circuit that is not reported. Since it is not reported there is no t' such that

$$V(7+, t') = v_1,$$
$$P(7+, t') = v_1, v_2, \ldots, v_p,$$
$$J(7+, t') = L[v_1].$$

Observing that Theorem 2 assures us that $E[L[v_i], L[v_1]] = T$ for $i = 1, 2, \ldots, p$ and using the properties of the breadth-first search algorithm we may now work backwards along the path to arrive at a contradiction; namely, there is no t" such that

$$P(7+, t'') = v_1,$$

which contradicts Theorem 2 and the implications of box 5 in Figure 4-4. ∎

Description of the flow diagram for the breadth-first search (BRFSRCH).

The flow diagram for the breadth-first search is shown in Figure 4-5. The variables V, I, P', J,and the array L, used in the second phase (Figure 4-4), are assumed to be available to this algorithm. A duplicate set of edge selectors, independent of those used in Figure 4-4, is used in this algorithm: the edge selector for vertex v is evaluated by SEL'(v) which operates like SEL(v) in

Figure 4-4. When execution of the breadth-first search begins, it is assumed that all edge selectors referenced by it are in the null state. Variables appearing in this flow diagram and not in Figure 4-4 are described below.

I': Pointer for P' indicating the tail vertex for edge selection.

V': Vertex considered for placement on P'.

A brief description of each box in the flow diagram follows:

0: Execution begins at this point.

1: The next vertex to be placed on P' is V ($V' \leftarrow V$), increment pointer for the head of the list P' ($I \leftarrow I + 1$), set pointer for tail vertex used in edge selection ($I' \leftarrow I$).

2: Put vertex on list P'.

3: Select an edge from vertex on P', the head vertex of this edge is assigned to V'.

4: Check if the edge selector for the vertex P' [I'] is in the null state, V' = 0 is TRUE if it is.

5: Does vertex V' have vertex number J? If the answer is yes this implies that there is a simple path from V back to the first vertex on P which does not cross the path on P.

6: If the pointers I' and I are equal, execution of this algorithm should terminate and no simple path from V back to the first vertex on P which does not cross the path on P exists.

7: Assign the value TRUE to BRFSRCH because the search was successful (cf. box 5).

8: Assign the value FALSE to BRFSRCH because the search was unsuccessful (cf. box 6).

9: Increment pointer to tail vertex for edge selection.

10: Is the vertex at the head of the selected edge already on P or on P'. If it is, then we will ignore it. If it is not, then we will put it on P'.

11: Increment pointer for the head of the list P'.

12: Execution terminates here.

Theorems and proofs for the breadth-first search algorithm (BRFSRCH). The algorithm BRFSRCH is called in the second phase to determine whether there is a path from vertex V (cf. Figure 4-4) to the first vertex on the path stack P, not containing any other vertex on P. If there is such a path then V lies on a circuit, hence it should be placed on P; in this case the TRUE exit from box 14, Figure 4-4, is taken, otherwise the FALSE exit is taken. These assertions from the substance of Theorem 7 and its corollarys. It is important to recognize that when BRFSRCH is called into execution the list P' may or may not be empty: it is empty when I = 0. It should be evident from the proof of Theorem 7 that if it is not empty then there is no path from any vertex on P' to the vertex numbered J, the first vertex on the stack P. Finally, it is to be observed that a test of the type implied by box 10, Figure 4-5, does not actually require a search of P'. We assume that an ordered list of the vertices on P' is maintained and determination of whether or not a vertex, v, is on P' simply requires interrogating this list at the position corresponding to v. Whenever I is set to zero in Figure 4-4 this list must be cleared. Maintenance of this list is not explicitly shown in the flow diagrams.

Lemma 14. Every vertex on P' is distinct from every other vertex on P' and from every vertex on P.

Proof of Lemma 14. Notice first that a vertex is placed on P' only by traversal of box 2. Second, notice that

$$V(1-, 0) \not\in P \wedge V(1-, 0) \not\in P'$$

is true; this follows from consideration of box 13 in Figure 4-4. Now suppose (2-, t') is necessary, then either (10F, t'-1) is necessary or (1-, t'-1) is necessary. In either case it is evident that

$$V'(2-, t') \not\in P \wedge V'(2-, t') \not\in P'$$

is true, hence every vertex placed on P' is distinct from every vertex on P' and from every vertex on P. Since P' is initially empty (implied by the operation $I \leftarrow 0$), the Lemma is true.∎

Lemma 15. For all feasible points (x, t), $t > 0$

$$I(x, t) \geq I'(x, t). \tag{4.9}$$

Proof of Lemma 15. Notice first that I' is changed only in boxes 1 and 9. Now (1+, 1) and (3-, 2) are necessary points and

$$I(1+, 1) = I'(1+, 1),$$

$$I(3-, 2) = I'(3-, 2),$$

hence (4.9) is true when $t = 1, 2$. Now either the algorithm terminates at $t = 6$ (by taking the path $3 \to 4 \to 5 \to 7$ or $3 \to 4 \to 6 \to 8$) or there will be a return to location 3- and

either

$$I(3-, 2 + \tau) = I'(3-, 2 + \tau),$$

or

$$I(3-, 2 + \tau) > I'(3-, 2 + \tau),$$

depending on whether or not box 11 was traversed (i.e., (11-, 6) is necessary). Notice that the point (6F, 5) is not feasible, hence I' could not have been incremented. Now if (3-, t') is feasible for some t' and

$$I(3-, t') \geq I'(3-, t'),$$

then either the algorithm terminates at t' + 4 or

$$I(3\text{-}, t' + \tau) \geq I'(3\text{-}, t' + \tau).$$

The truth of this is evident from the observation that if equality holds at (3-, t') then (9-, t' + 3) is not feasible; and if equality does not hold then I' can only be incremented by 1. Thus (4.9) is true for all feasible (3-, t). Finally, consideration of possible paths from (3-, t') to (3-, t' + τ) or (12-, t' + 4) easily shows (4.9) is true for all feasible (x, t). ∎

Lemma 16. Algorithm BRFSRCH terminates.

Proof of Lemma 16. Box 2 can only be traversed a finite number of times since there are a finite number of vertices in the graph and by Lemma 14 all vertices on P' are distinct. Consequently box 11 can only be traversed a finite number of times, thus I is bounded. Since, by Lemma 15, I' (x, t) $\leq$ I(x, t) it follows that box 9 can only be traversed a finite number of times. Thus if the algorithm did not terminate it would be necessary to traverse the path 3 → 4 → 5 → 10 → 3 an infinite number of times. But this is impossible because SEL' must eventually return a value of zero since the graph contains only a finite number of vertices. ∎

Theorem 7. Suppose

$$P(1\text{-}, 0) = v_1, v_2, \ldots, v_k,$$
$$V(1\text{-}, 0) = v,$$
$$J(1\text{-}, 0) = j, \qquad (L[v_1] = j),$$
$$I(1\text{-}, 0) = 0,$$

then for some t' > 0 one of the following is true:

(a) BRFSRCH (12-, t') = T;

(b) BRFSRCH (12-, t') = F.

Moreover (a) is true iff there is a simple path from v to v_1 which does not include any of the vertices v_2, v_3, ..., v_k; otherwise (b) is true.

Proof of Theorem 7. Since, by Lemma 16, the algorithm terminates and termination implies traversal of box 9 or box 10, it follows that BRFSRCH will be assigned the value T or F before termination. At (2+, 2) we have

$$P'[1](2+, 2) = v,$$

$$I(2+, 2) = 1.$$

It is evident from a consideration of the path $3 \to 4 \to 5 \to 10 \to 11 \to 2 \to 3$ that if another vertex is placed on P' then there must be an edge from v to that vertex. By carrying out an obvious induction it is evident that there is a path from v to every vertex placed on P' and this path consists only of vertices on P'. Now suppose that (7+, t') is necessary (i.e., the algorithm terminates with BRFSRCH equal to T) then (5T, t'-1) is necessary hence there must be an edge from a vertex on P' to the vertex v_1, so there must be a path from v through vertices only on P' to v_1. Since the vertices on P' are distinct from those on P by Lemma 14 it follows that if (a) is true then there is a path from v to v_1 which does not include any of the vertices v_2, v_3, ..., v_k.

We now need only prove the converse, namely that if there is a path from v to v_1 which does not include any of the vertices v_2, v_3, ..., v_k then (a) is true. The proof is by contradiction. Suppose there is such a path, say v_1, v_2', v_3', ..., v_m', v_1 where $v_i' \neq v_j'$ for all $i \neq j$ with $2 \leq i, j \leq m$ and (b) is true. We must conclude v_m' was never placed on P' (i.e., for no t is $V'(3+, t) = v_m'$) otherwise at a later time, t', the edge

(v'_m, v_1) would be selected and (5T, t') would be necessary. A similar argument leads us to the conclusion that $v'_{m-1}, \ldots v'_3, v'_2$, v could not have been placed on P'. Since v was placed on P' we arrive at a contradiction consequently under these hypotheses (b) cannot be true.∎

Corollary 1. Suppose

$$P(1-, 0) = v_1, v_2, \ldots, v_k,$$
$$V(1-, 0) = v,$$
$$J(1-, 0) = j, \qquad (L[v_1] = j),$$
$$I(1-, 0) = i,$$

then for some t' > 0 one of the following is true:

(a) BRFSRCH(12-, t') = T;

(b) BRFSRCH(12-, t') = F.

Moreover (a) is true iff there is a simple path from v to v_1 which does not include any of the vertices $v_2, v_3, \ldots, v_k$; otherwise (b) is true.

Proof. Consideration of boxes 10 and 15 in Figure 4-4 shows that if $i \neq 0$ then P cannot have changed since the last execution of BRFSRCH. Moreover the last execution of BRFSRCH must have assigned the value FALSE to BRFSRCH. Assume that I(1-, 0) = 0 on the last execution of BRFSRCH, then Theorem 7 applies and since this execution assigned the value FALSE to BRFSRCH it follows that there is no path from any vertex on P' to v_1. Now an argument like that used in proving Theorem 7 can be applied to prove this Corollary. If $I(1-, 0) \neq 0$ on the last execution of BRFSRCH, then an obvious induction argument leads to the result stated in this Corollary.∎

Corollary 2. Suppose

$$P'(12-, t') = v'_1, v'_2, \ldots, v'_k$$
$$BRFSRCH(12-, t') = F$$
$$J(12-, t') = j$$

then there is no path from any of the vertices v_i' to the vertex numbered j which does not intersect a vertex on P or P'.

This Corollary is an immediate consequence of Corollary 1.

5. EFFICIENCY

Any attempt to discuss the efficiency of circuit finding algorithms is, of necessity, hampered by the lack of universally accepted efficiency measures. The efficiency of sorting algorithms is generally measured in terms of the number of comparisons required. The efficiency of matrix computation algorithms is usually measured in terms of the number of multiplications involved. No such clear-cut yardsticks exist in our area.

It seems to us that in order for such a measure to be of value it must somehow reflect not only execution speed but also storage requirements. We are convinced that our algorithm is like most others in that given more storage it could be altered to execute faster. Thus, in order to give a balanced picture of our algorithm's efficiency, we will first discuss its storage and then its execution speed.

The storage requirements of this algorithm are roughly proportional to the square of n, the number of vertices in the graph. Two structures account for nearly all of the storage utilization -- the E array and the representation of the actual graph. The graph is represented by n circularly linked lists, where the ith list contains representations of each of the vertices reachable as a destination vertex from vertex i. Each of the n lists also contains a list header. Thus, a graph having e edges requires n + e list nodes for its representation. Since a graph on n vertices may have up to n^2 edges, a graph on n vertices will require at most $n + n^2$ list nodes. Hence, the storage requirement for the graph representation is at most roughly proportional to n^2.

The E array is, as noted before, a lower triangular Boolean array. Since each vertex has a row in E it is clear that the size of E is proportional to

$\frac{(n+1)n}{2} = \frac{1}{2}(n^2 + n)$. Hence the storage requirement for E is also roughly proportional to n^2.

The algorithm also makes use of various other arrays. These, however, are all linear in n. Hence it seems fair to claim that the algorithm requires roughly $k_1 n^2$ list nodes and roughly $k_2 n^2$ Boolean switches where k_1, k_2 are constants. Thus it requires storage roughly proportional to n^2.

It is in trying to assess the execution speed of the algorithm that we are most hampered by the lack of agreed upon measures.

Tiernan [1] claims that his algorithm is "the theoretically most efficient," observing that each circuit of the graph is considered only once. He claims that each circuit must be considered at most once and therefore, that his algorithm is "most efficient." It is not hard to find cases in which Tiernan's procedure seems to be highly inefficient. A good example is found in Tarjan [3].

An appropriate measure of execution time efficiency seems to be the number of edge traversals made during the execution of a graph searching algorithm. By an edge traversal we mean the process of getting to a head vertex from a given tail vertex. The traversal itself requires that we access the structure used to represent the graph, and the procedure followed upon reaching a head vertex seems to involve some more or less standard sequence of tests, followed by another edge traversal. Hence the edge traversal would seem to be an appropriate unit of work.

It is clearly unsatisfactory to compute the number of edge traversals required every time we consider a new graph, however. A more useful approach is to produce an upper bound for the number of edge traversals required which

is valid over a wide class of graphs. Among those who have addressed this problem, Hopcroft and Tarjan [4] have adopted what seems to us to be a useful, but sometimes misleading scheme. They produce "order of magnitude" estimates for the maximum amount of work required to process all graphs on n vertices and e edges, casting these estimates as functions of n and e. Hence they may, for instance, describe the efficiency (or complexity) of a graph algorithm to be $O(n + e)$, (order of $n + e$). A rigorous definition of this terminology might be the following:

> Let A be a graph algorithm whose complexity is $O(n + e)$. Then if G is a graph on n vertices having e edges, there exists a constant K such that A will process G in at most $K \times (n + e)$ operations, the operations themselves being independent of n and e.

Since we feel that such estimates are useful we now develop a complexity estimate for our algorithm. During phase 1 our algorithm traverses each edge of the graph exactly once. We omit the detailed proof of this assertion but one can see the truth of it fairly easily by observing that a vertex is removed from P after every edge from it has been selected once (cf. boxes 4 and 5 of Figure 4-2) and after it has been removed from P it cannot again be placed on P (cf. box 8 of Figure 8). Each such traversal will necessitate at most a copy operation (if the head vertex is marked old). Additionally, whenever a vertex is popped off the stack, a COPY* operation must be performed (i.e., a COPY operation for up to n - 1 rows). Hence, considering a COPY operation to be $O(n)$ in complexity, we easily see that the complexity of phase 1 is

$$O(e \times n + n^3).$$

Since $e \leq n^2$ the above is equivalent to

$$O(n^3).$$

Phase 2 is arranged so that no vertex will be placed on the search stack unless it is known that the vertex is part of an evolving circuit. We observe that a circuit can have at most n vertices. Thus, the complexity per circuit in

phase 2 can be easily determined from the amount of work done between placing two vertices on the search stack.

Having just placed a vertex on the search stack in phase 2, we recall that we next examine in turn each head vertex reachable from the newly stacked vertex. In the worst case, it will be necessary to execute a breadth-first search from each of these head vertices before being able to stack one. It is clear that no edge of the graph will be traversed more than once during any breadth-first search. But, additionally, we readily see that even in this worst case any edge traversed in a breadth first search from one particular vertex will not be traversed during the breadth first search of any of the other vertices. Hence the algorithm will perform at most e edge traversals before finally stacking the next vertex.

From this it is clear that the complexity of phase 2 is

$$O(c[n \times e]),$$

where c is the number of circuits. Thus the complexity of the entire algorithm is

$$O(n^3 + c[n \times e]). \tag{5-1}$$

In many graphs c is quite large in comparison to n and e. At worst, c may grow factorially in n since there are $(n - 1)!$ n-circuits in the complete graph on n points. Thus it is not unreasonable to observe that the second term of the complexity estimate often dominates the first. Hence, regarding phase 1 as a preliminary phase which contributes a fixed overhead cost of $O(n^3)$, we see that the dominant cost of the algorithm is

$$O(c[n \times e]).$$

Although the efficiency estimate, (5-1), is an interesting and important one, we would at this point like to observe that it can also be misleading.

We can easily construct an algorithm similar to the one presented here by substituting a vector, E*, for our E array and defining E* by:

$$E^*[i] = E[i, i] \quad 1 \leq i \leq n.$$

In our new algorithm we would test E*[i] before embarking upon a search for cycles of class i, but during the search itself, we would rely solely upon the breadth-first search method described here. The efficiency estimate of this simple algorithm is O(e + c[e×n]). It is particularly interesting that the complexity of phase 2 of this algorithm is O(c[n×e]) which is identical to the complexity of phase 2 of our original algorithm. It seems clear, however, that the original algorithm would execute phase 2 more rapidly and in many cases, offer very large improvements in speed.

On the face of it, it might appear that the simpler algorithm is the preferable one because it reduces the amount of intermediate storage required while also lowering the estimate of execution time. Yet this example shows why inordinate reliance upon such estimates can be misleading. The extra storage required for our algorithm can decrease actual running times considerably while nevertheless failing to decrease the execution time estimate, as represented by the complexity figure.

6. REFERENCES

1. Tiernan, James C. An efficient search algorithm to find the elementary circuits of a graph. Comm. ACM 13, 12 (Dec. 1970), 722-726.

2. Weinblatt, Herbert. A new search algorithm for finding the simple cycles of a finite directed graph. JACM 19, 1 (Jan. 1972), 43-56.

3. Tarjan, Robert. Enumeration of the elementary circuits of a directed graph. TR 72-145, Department of Computer Science, Cornell University (Sept. 1972), 1-13.

4. Hopcroft, John and Tarjan, Robert. Efficient algorithms for graph manipulation. Comm. ACM 16, 6 (June 1973), 372-378.

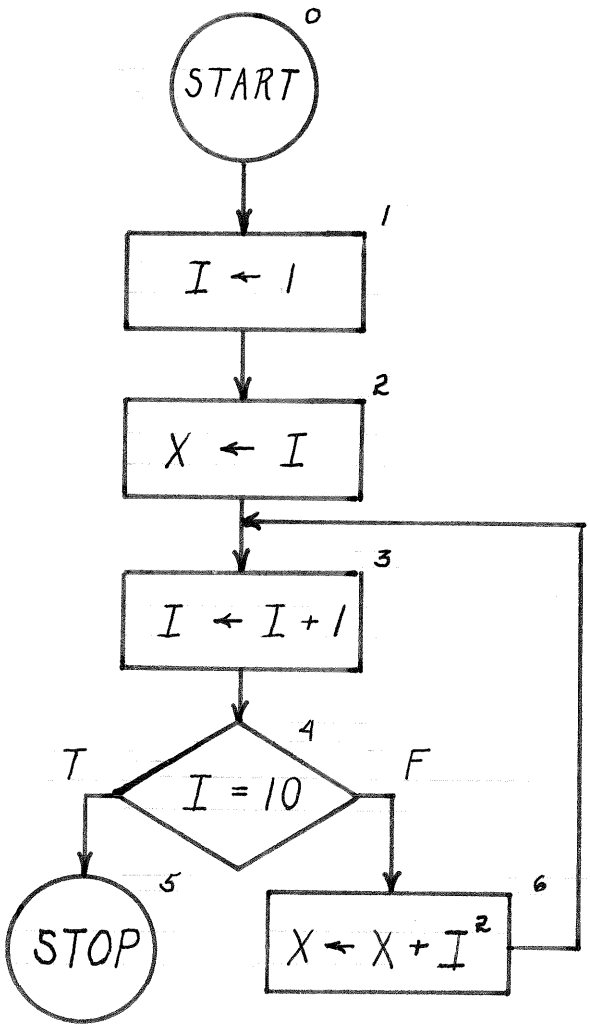


Figure 4-1: Illustration of flow diagram notation for informal proof technique

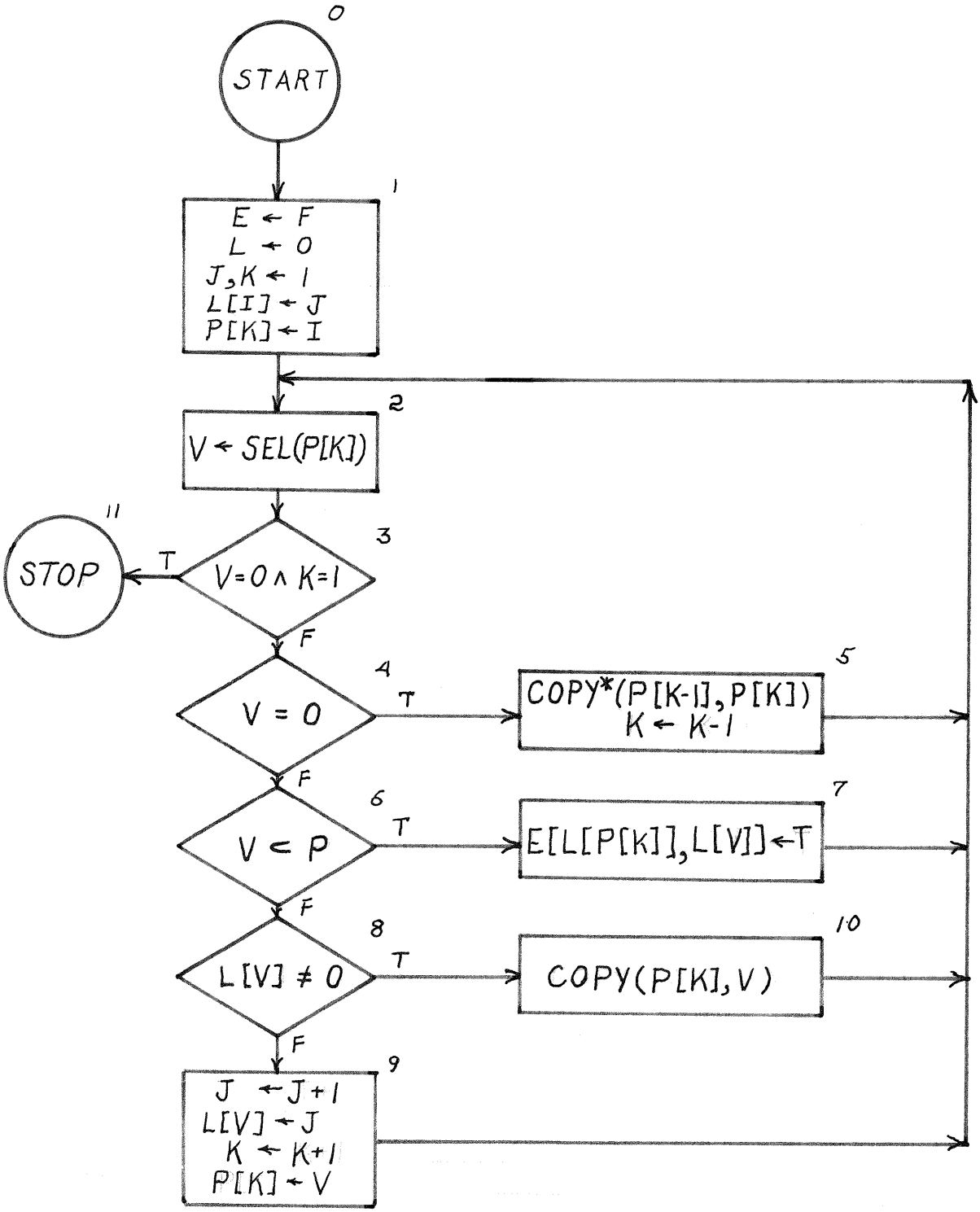


Figure 4-2: Flow diagram for the first phase

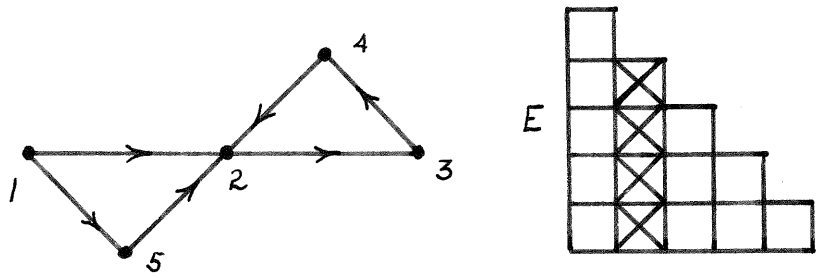
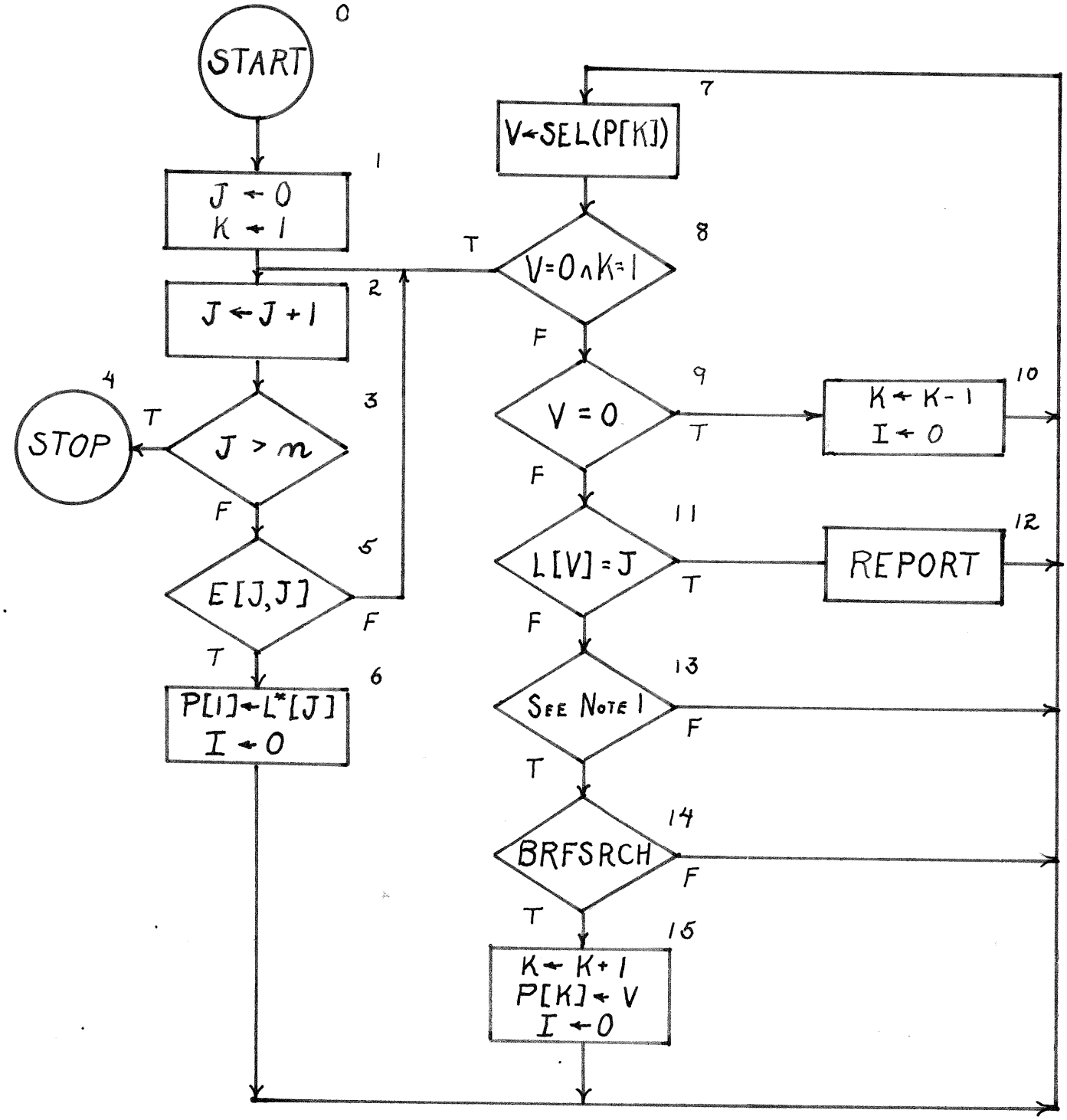


Figure 4-3

Example showing that the converse of Theorem 2 is false. Vertex numbers and the E array are shown here when the first phase terminates. Notice that E[5, 2] is true but the vertex numbered 5 is not on a circuit of class 2.



Note 1 : $L[V] > J \wedge E[L[V], J] \wedge V \notin P \wedge V \notin P'$

Figure 4-4: Flow diagram for the second phase

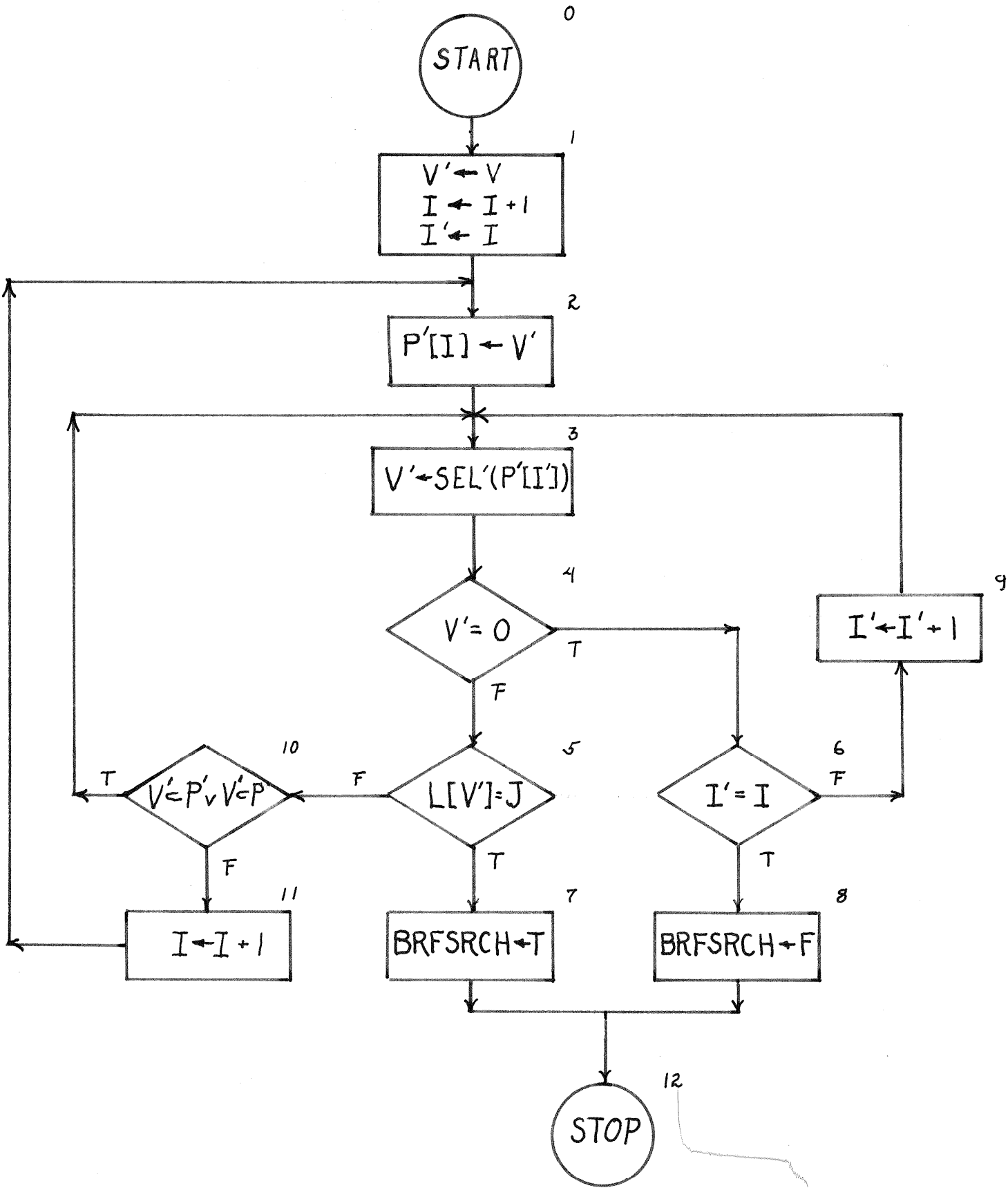
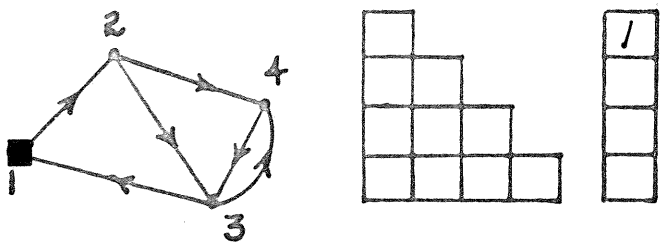
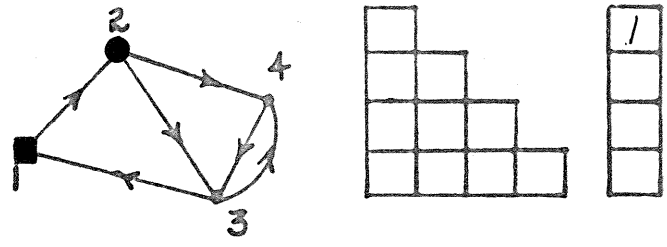


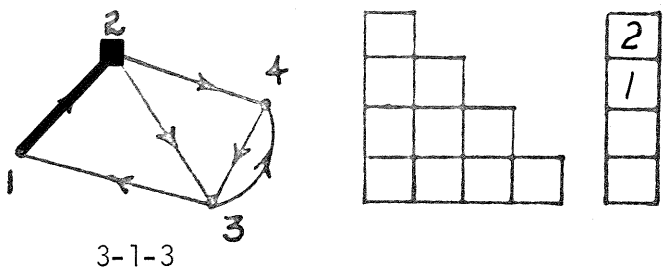
Figure 4-5: Breadth-first search algorithm (BRFSRCH)



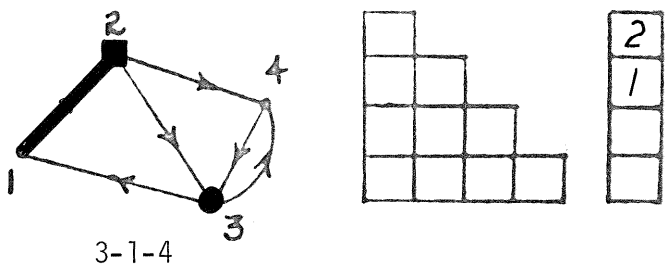
3-1-1



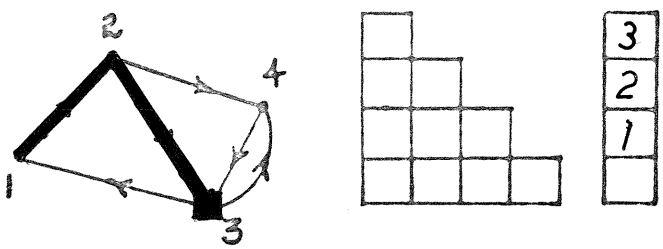
3-1-2



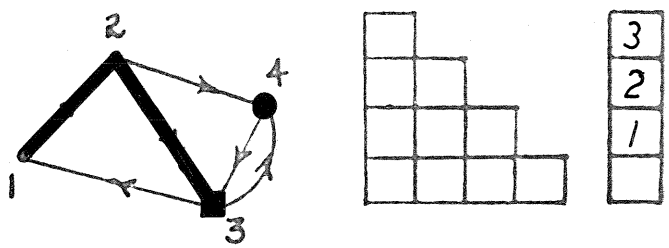
3-1-3



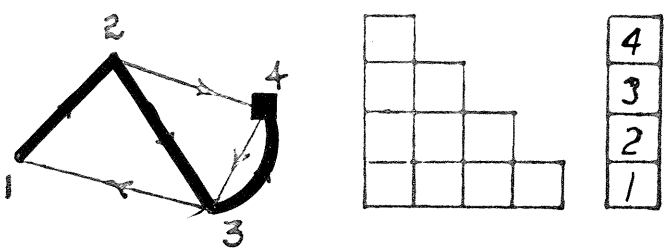
3-1-4



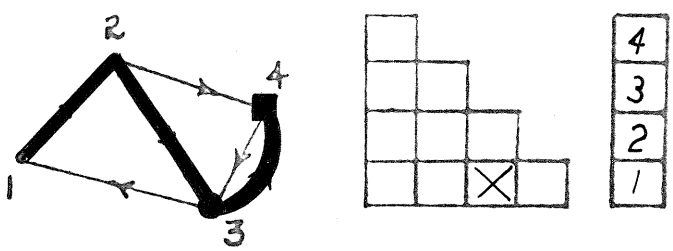
3-1-5



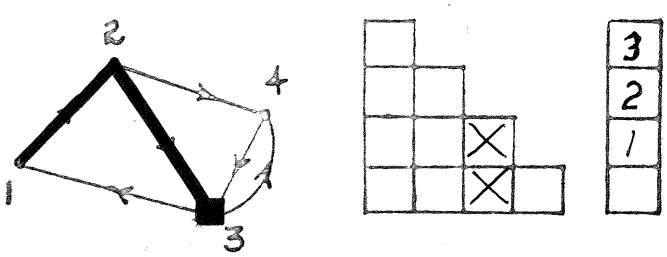
3-1-6



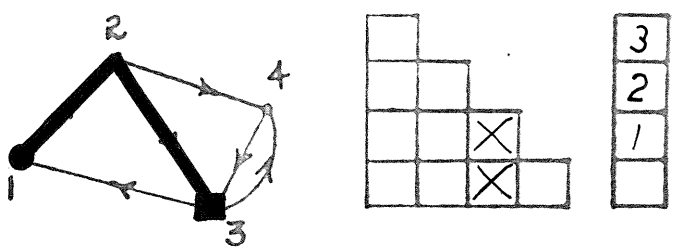
3-1-7



3-1-8

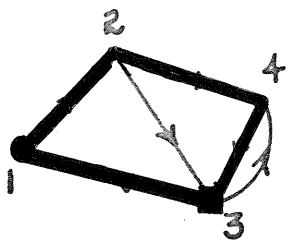


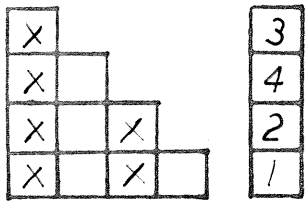
3-1-9



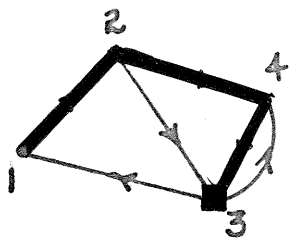
3-1-10

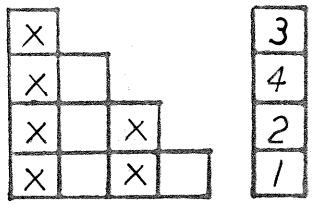
FIGS 3-1-1 THROUGH 3-1-10



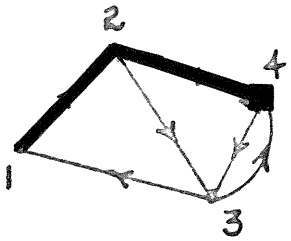


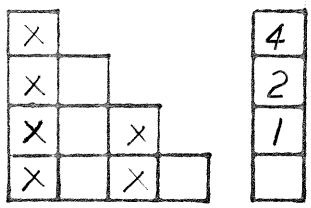
3-1-31



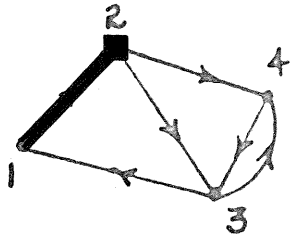


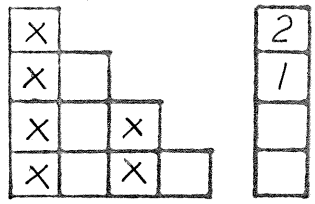
3-1-32



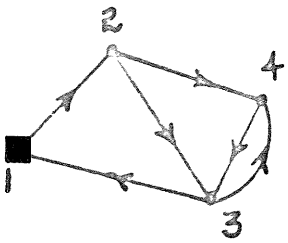


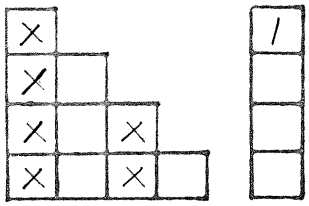
3-1-33



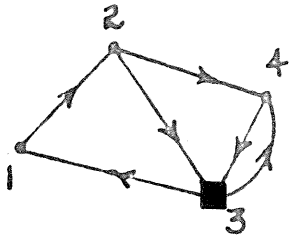


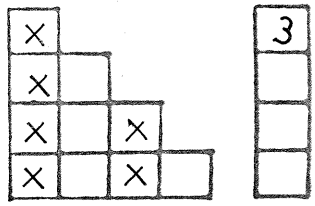
3-1-34



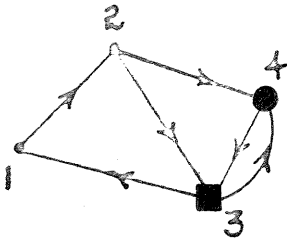


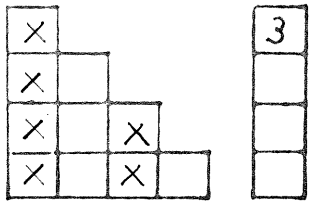
3-1-35



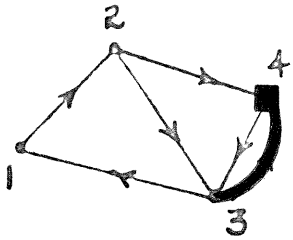


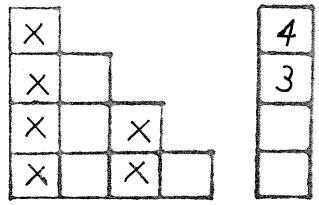
3-1-36



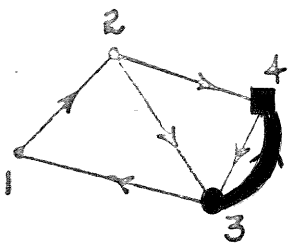


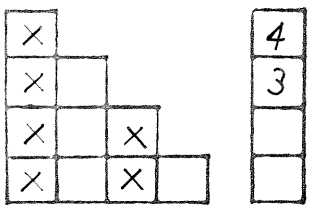
3-1-37



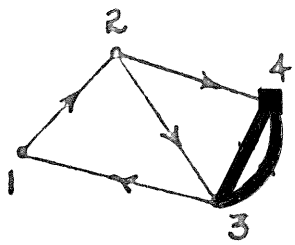


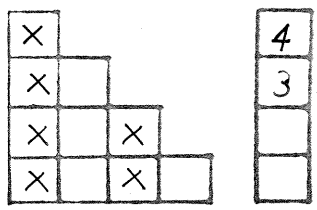
3-1-38





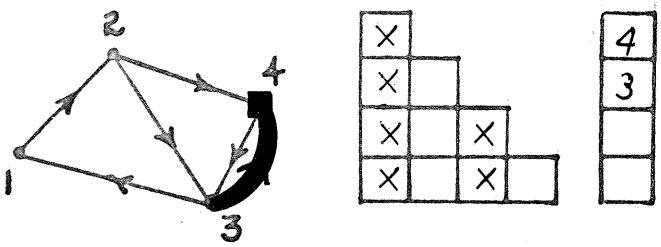
3-1-39



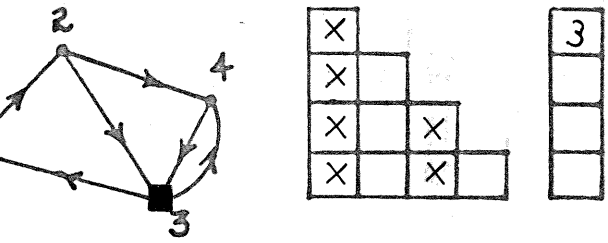


3-1-40

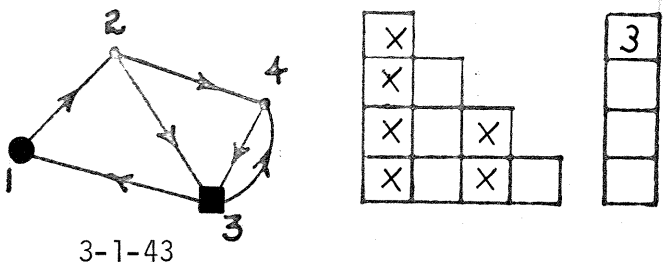
FIGS 3-1-31 THROUGH 3-1-40



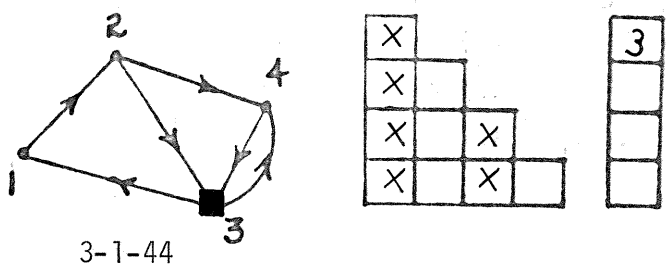
3-1-41



3-1-42



3-1-43



3-1-44

FIGS 3-1-41 THROUGH 3-1-44