The Expression of Algorithms by Charts — **Source link** ⬈

John L. Bruno, Kenneth Steiglitz

Institutions: Pennsylvania State University, Princeton University

Related papers:

- Flow diagrams, turing machines and languages with only two formation rules

- Notes on avoiding "go to" statements

- On the capabilities of while, repeat, and exit statements

- Letters to the editor: go to statement considered harmful

- Analysis of structured programs

The Expression of Algorithms by Charts

J. BRUNO AND K. STEIGLITZ

Princeton University, Princeton, New Jersey

ABSTRACT. This paper discusses the expression of algorithms by flowcharts, and in particular by flowcharts without explicit go-to's (*D*-charts). For this purpose we introduce a machine independent definition of algorithm which is broader than usual. Our conclusion is that *D*-charts are in one technical sense more restrictive than general flowcharts, but not if one allows the introduction of additional variables which represent a history of control flow.

KEY WORDS AND PHRASES: flowcharts, go-to-less programming, *D*-charts, algorithm expression

CR CATEGORIES: 5.20, 5.24

1. *Introduction*

The term "algorithm" is used in many different ways. Sometimes we speak of an algorithm as a process in the abstract, without reference to a particular computer. It is in this sense, for example, that we speak of the "radix exchange sort algorithm," or the "simplex algorithm." Often we identify an algorithm with a particular sequence of instructions for a particular computer.

In this paper we shall present a new definition of algorithm which emphasizes the sequence of commands associated with a particular "input." We then define the notion "expression" of algorithms by general flowcharts and flowcharts without explicit go-to's (*D*-charts). Some theorems are given which exhibit some of the relationships between algorithms, flowcharts, and *D*-charts.

2. *Algorithms*

Central to our discussion is the notion of an algorithm which is defined independently of its expression in a given language. One such definition of an algorithm can be given as follows:

Let N be a set of *variables* or *names*. If $n \in N$, then n takes on *values* in a *value set* V_n. Let C be a finite set of "sufficiently basic" operations called *commands*. All members of C are of the form $y \leftarrow f(y_1, \cdots, y_k)$, where $k \geq 0$, $y, y_1, \cdots, y_k$ are members of N, and f is some function of the values of the names $y_1, \cdots, y_k$. A function s which associates with each member of N a value in the corresponding value set is called a *state function*; that is, if for every $n \in N$, $s(n) \in V_n$, then s

is called a state function. Let S denote a prechosen class of state functions called *initial state functions*. An *algorithm* A is a function which associates with each member $s \in S$ a finite sequence $A(s)$ of members of C (possibly the null sequence λ). *The execution of A with respect to $s \in S$* is a finite sequence $s_0 s_1 \cdots s_u$ of state functions determined by:

1. $s_0 = s$.

2. Suppose $y \leftarrow f(y_1, \cdots, y_k)$ is the ith term in the sequence $A(s)$; then $s_i(n) = s_{i-1}(n)$ for all $n \in (N - \{y\})$ and $s_i(y) = f(s_{i-1}(y_1), \cdots, s_{i-1}(y_k))$.

3. u is equal to the number of terms in $A(s)$.

Example: Algorithm SEARCH. $N = \{M, \text{list}[\ ], \text{count}[\ ], i, x\}$. We are using list[ ] (and count[ ]) to denote a countably infinite number of names; specifically list[ ] is shorthand for the names list[1], list[2], list[3], $\cdots$. $S = \{s \mid s(M) \geq 1\}$. This algorithm searches list[1], $\cdots$, list[M] for x. If it is found at position j, say, count[j] is incremented by 1. If it is not found, it is appended to the list at position $M + 1$, count[$M + 1$] is initialized at 1, and finally M is incremented by 1.

Let $C = \{c_1, \cdots, c_6\}$.

$$c_1 \triangleq i \leftarrow 1, \qquad\qquad c_4 \triangleq \text{list}[i] \leftarrow x,$$

$$c_2 \triangleq i \leftarrow i + 1, \qquad\qquad c_5 \triangleq \text{count}[i] \leftarrow 1,$$

$$c_3 \triangleq \text{count}[i] \leftarrow \text{count}[i] + 1, \qquad c_6 \triangleq M \leftarrow i.$$

SEARCH is given by

$$\text{SEARCH}(s) = \begin{cases} c_1(c_2)^{j-1}c_3 & \text{if } s(x) = s(\text{list}[j]), \ 1 \leq j \leq s(M), j \text{ is as small} \\ & \text{as possible,} \\ c_1(c_2)^{s(M)}c_4c_5c_6 & \text{if } s(x) \neq s(\text{list}[j]), \ 1 \leq j \leq s(M). \end{cases}$$

The above definition of an algorithm employs only the sequence of commands to be carried out and says nothing about how one determines the appropriate command sequence for each initial state function. This allows us to discuss the idea of having more than one expression for a given algorithm. Our primary concern is with the finite expression of algorithms by charts which indicate in a schematic way the "flow of control" from command to command.

3. Flowcharts and D-Charts

Clearly, if there are only a finite number of allowable initial state functions one could simply catalog the appropriate command sequences of an algorithm. Complications arise when there is an infinite number of possible initial state functions. We shall use a special class of flowcharts called D-charts as a means of expressing algorithms.

By a *flowchart* F we mean a finite directed graph which satisfies the following:

1. Each of the vertices of F must be one of the following types:

   (a) *Start vertex*:

   F contains precisely one start vertex, and this vertex has exactly one edge incident away from it and no edges incident toward it.

   (b) *Stop vertex*:

F contains precisely one stop vertex, and this vertex has one or more edges incident toward it and no edges incident away from it.

(c) *Command type*:

α is a sequence of commands; there are one or more edges incident toward a command vertex; exactly one edge is incident away.

(d) *Decision type*:

There are two edges leaving a decision vertex; one of these edges is labeled with a *quantifier-free* predicate P and the other with $\sim P$, the negation of P; there are one or more edges incident toward a decision vertex.

2. *F* is a connected graph (in the undirected sense).

We consider a quantifier-free predicate to consist of "atoms" which are combined according to the rules of the propositional calculus. The *atoms* are relations of the form $R(y_1, \cdots, y_k)$ where $y_i \in N$ for $i = 1, \cdots, k; k \geq 0$; and R takes on the value "true" or "false" when we substitute the values of the variables y_i in $R(y_1, \cdots, y_k)$.

D-charts (after Dijkstra [1]) are a restricted class of flowcharts defined recursively by the following grammar:
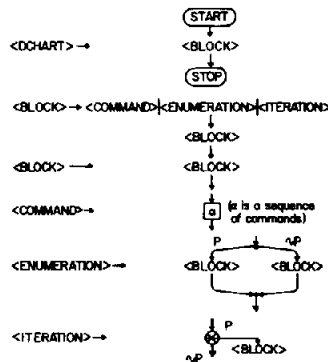
Note that the following conventions have been used:

(1) in the ⟨ENUMERATION⟩ rule we have used $\bigvee$ for $\underset{\text{A}}{\bigcup}$, and

(2) in the ⟨ITERATION⟩ rule we have used
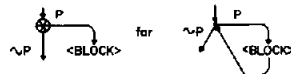
It should be clear that this last convention causes no difficulty in determining where to return after we have "executed" ⟨BLOCK⟩.

The quantities defined for algorithms in Section 2 are defined analogously for flowcharts; thus we may speak of the set N of variables of a flowchart F, the set S

of initial state functions of F, and the set C of commands associated with F. The sequence of commands and state functions associated with each $s \in S$ is determined by F as follows: *initially*, we have $s \in S$ as our current state function, a command sequence c equal to λ, and a state function sequence $\sigma = s$, and we are positioned at the START vertex of F.

Suppose we are at a vertex $\nu \subset F$, with the current state function s', current command sequence c, and current state function sequence σ. We shall describe how one determines a new current state function, updates c and σ, and chooses a new current vertex in F:

(1) If ν is the START vertex, s', c, and σ are unchanged and we move to the unique successor of the START vertex.

(2) If ν is a command vertex and $\alpha = c_1 c_2 \cdots c_m$, then c becomes $c c_1 c_2 \cdots c_m$, s' becomes s_m, and σ becomes $\sigma s_1 \cdots s_m$; where if $c_i \equiv y \leftarrow f(y_1, \cdots, y_k)$, then $s_i(n) = s_{i-1}(n)$ for all $n \in N - \{y\}$ and $s_i(y) = f(s_{i-1}(y_1), \cdots, s_{i-1}(y_k))$, and where $s_0 = s'$. We move to the unique successor of the command vertex.

(3) If ν is a decision vertex, s', c, and σ remain unchanged. We evaluate P with

A D-chart which expresses Algorithm SEARCH:
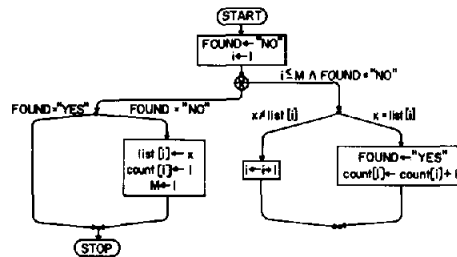
D

A flowchart which directly expresses Algorithm SEARCH:
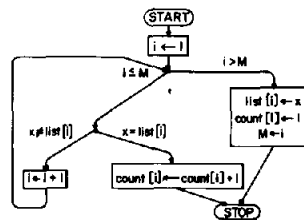
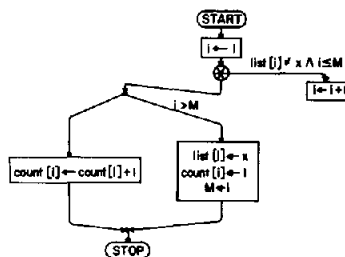F

A D-chart which directly expresses Algorithm SEARCH:

Fig. 1

respect to s' and if P is "true" we move to the successor of ν determined by the edge labeled P, otherwise we move to the successor of ν determined by the edge labeled $\sim P$.

(4) If ν is the STOP vertex we define $F(s) = c$ and define the *execution of F with respect to s* to be σ.

Let F be a flowchart and let N', V_n' for each $n \subset N'$, C', and S' be associated with F. Let A be an algorithm and the quantities N, V_n for each $n \in N$, C, and S be associated with A. We say that F *directly expresses* algorithm A if:

1. $N = N'$, $V_n = V_n'$ for each $n \in N$, $C = C'$, $S = S'$, and
2. for each $s \in S$ $F(s) = A(s)$.

Let $\epsilon_d(A)$ denote the set of flowcharts which directly express algorithm A.

Direct expression of an algorithm does not always provide easily understood flowcharts, and accordingly we say that F *expresses* algorithm A if:

1. $N \subseteq N'$, $V_n = V_n'$ for $n \in N$, $C \subseteq C'$, $S = S'/N$ (S'/N denotes the set of functions obtained by restricting the members of S' to the set N), and
2. for each $s \in S$, $A(s)$ is a subsequence of $F(s')$ where s' is any member of S' whose restriction to N is equal to s, and the state functions in the execution of A with respect to s are equal to the restrictions to N of the corresponding state functions in the execution of F with respect to s'.

Let $\epsilon(A)$ denote the set of flowcharts which express algorithm A. Since flowcharts are necessarily finite, it may be that $\epsilon(A) = \phi$. Furthermore, $\epsilon_d(A) \subseteq \epsilon(A)$. See Figure 1.

D-charts are as "powerful" as flowcharts in the following sense:

THEOREM 1 (Böhm and Jacopini [2]). *If $F \in \epsilon(A)$ then there exists a D-chart $D \in \epsilon(A)$.*

PROOF. Label the START vertex in F with n_0 and the STOP vertex with n_∞, and label all other vertices of F with the labels $n_1, \cdots, n_m$. If n_i is a command vertex we construct a corresponding D-chart block as follows:
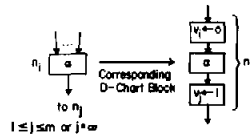
If n_i is a decision vertex we construct a corresponding D-chart block as follows:
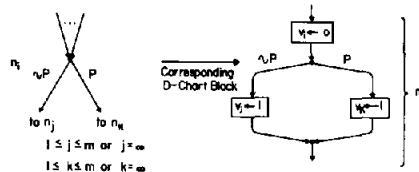
If N is the set of variables of F we assume that $v_1, \cdots, v_m, v_\alpha$ are not in N and that these new variables take on values in $\{0, 1\}$. If n_i' is the D-chart block corresponding to vertex n_i in F, we construct the D-chart shown in Figure 2, where we assume that n_j is the vertex in F which directly succeeds the START vertex ($1 \leq j \leq m$ or $j = \infty$) and $m \geq 1$ (if $m = 0$ then $D = F$). It is easy to see that $D \in \epsilon(A)$. ∎

Suppose $F \in \epsilon_d(A)$ and R is the set of all atoms used in the formation of predicates associated with F. We say that a flowchart F' is *directly equivalent* to F
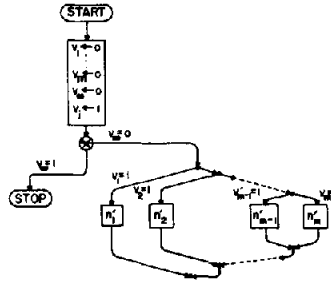
FIG. 2

with respect to A and the atoms R of F, if $F' \in \epsilon_d(A)$ and the predicates appearing in F' are formed using only atoms in R.

The following theorem is analogous to a theorem of Knuth and Floyd [3].

THEOREM 2.   *There exists an algorithm A and a flowchart $F \in \epsilon_d(A)$ such that there is no D-chart which is directly equivalent to F with respect to A and the atoms R of F.*

PROOF.   Consider the following algorithm A and flowchart F where $F \in \epsilon_d(A)$.

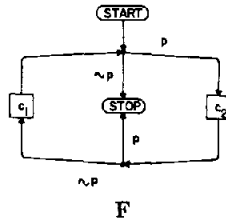$$N = \{M, i\}, \quad V_M = V_i = \{1, 2, \cdots\},$$

$$S = \{s \mid s(M) \geq 1, \; s(i) = 1\}, \quad C = \{c_1, c_2\}, \quad R = \{p\},$$

$$c_1 \triangleq i \leftarrow 2M - i,$$

$$c_2 \triangleq i \leftarrow 2M - 2 - i,$$

$$p \triangleq i < M,$$

$$A(s) = \begin{cases} (c_2 c_1)^k; & s(M) \text{ odd and } \quad k = [s(M) - 1]/2, \\ c_2 (c_1 c_2)^h; & s(M) \text{ even and } \quad h = [s(M) - 2]/2. \end{cases}$$

F

| M	Sequence of values of i
1	$(\sim p)1(\sim p)$
2	$(p)1(p)1(p)$
3	$(p)1(p)3(\sim p)3(\sim p)$
4	$(p)1(p)5(\sim p)3(p)3(p)$
5	$(p)1(p)7(\sim p)3(p)5(\sim p)5(\sim p)$
6	$(p)1(p)9(\sim p)3(p)7(\sim p)5(p)5(p)$
7	$(p)1(p)11(\sim p)3(p)9(\sim p)5(p)7(\sim p)7(\sim p)$

The predicates in parentheses hold at their respective points in the sequence Consider the following sequence of commands and predicates

$$\alpha = (p)c_2(\sim p)c_1(p)c_2(\sim p)c_1(p)c_2(\sim p)c_1(p)c_2(\sim p) \cdots.$$

We interpret the above sequence as a metadescription of the execution of F for arbitrarily large M. Specifically, the subsequence of commands is the algorithm $A(s)$ for arbitrarily large $s(M)$; the predicate following each command holds after the corresponding command is executed, that is, (p) following c_1 means that $i < M$ after c_1 is executed, and $(\sim p)$ following c_2 means that $i \geq M$ after c_2 is executed.

We make the assumption that there is a D-chart D which is directly equivalent to F, and we shall show that this leads to a contradiction. Let us "follow" the execution of $D(s)$ when $s(M)$ is arbitrarily large; suppose I is the first vertex in D which we visit for a second time. By the structure of D, I is an iteration vertex
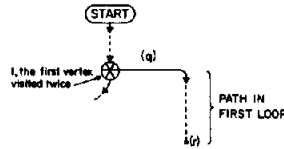

The symbol (q) represents the atom (either p or $\sim p$) which was true when this loop was entered for the first time, and (r) is the atom which was true when the vertex I was reached for the second time.

Assume that $(q) = (p)$ and $(r) = (\sim p)$. By inspection of α it is clear that $\boxed{c_2}$ must appear on the path
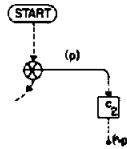
One can choose $s'(M)$ such that $D(s')$ behaves exactly like $D(s)$ until $\boxed{c_2}$ is reached for the first time, at which time in the execution of $D(s')$, c_2 causes p to be true and this is in fact the last command executed for this value of $s'(M)$. This results in infinite looping and consequently $(q) = (p)$ and $(r) = (\sim p)$ cannot hold. A similar argument shows that $(q) = (\sim p)$ and $(r) = (p)$ cannot hold.

Assume that $(q) = (p)$ and $(r) = (p)$.
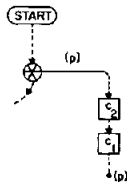
In this case $\boxed{c_2}$ and $\boxed{c_1}$ must appear at least one time as consecutive command vertices in the loop. Again by choosing $s'(M)$ properly we can cause p to hold after c_2 is executed for the first time in this loop. This again implies infinite looping and consequently $(g) = (p)$ and $(r) = (p)$ cannot hold. Similarly, $(g) = (\sim p)$ and $(r) = (\sim p)$ cannot hold. We are therefore forced to conclude that there is no repeated vertex in the execution of $D(s)$ when $s(M)$ is arbitrarily large. This is im-

possible since D is finite and we can conclude that there is no D-chart which is directly equivalent to F. ∎

4. *Flags*

In the proof of Theorem 1 we introduced the variables v_i in order to construct the appropriate D-chart, as did Böhm and Jacopini [2]. We can think of these variables as "flags" or "signals" which tell us which sequence of commands to execute next. We can make the notion of a "flag" more precise. Let N be the set of variables associated with a flowchart F. We say that a variable $x \in N$ is a *flag* if x takes on values in a finite set, and in each command of the form $x \leftarrow f(y_1 \cdots y_m)$, each of the variables $y_1, \cdots, y_m$ are flags. The variable FOUND in the D-chart which expressed Algorithm SEARCH is a flag.

Intuitively one would think that flags are unessential in a flowchart, and in fact it is easy to show that they are dispensible in a certain sense. For example, suppose we wanted to eliminate the flag FOUND in the Example. Since FOUND takes on only two values we can make two copies of D (considered as a flowchart), one with the value of FOUND considered to be "NO" and the other with FOUND set to the value "YES." Any statement which changes the value of FOUND is replaced by an appropriate transfer.

Figure 3 shows two copies of D, one for each of the possible values of FOUND. The dotted edges are edges which have been omitted. A single START vertex has been added, and it is immediately followed by a test to determine the appropriate copy of D to begin with. Figure 3 can be reduced to a flowchart by successively eliminating all vertices with no incoming edges, replacing all series edges by single edges, replacing all parallel edges by single edges, and finally coalescing all the STOP vertices into a single STOP vertex. This procedure applied to Figure 3 results in the flowchart F in the Example.

We say that a set W of flags is *complete* if $x \leftarrow f(y_1, \cdots, y_k) \in C$ and $x \in W$ imply $y_1, \cdots, y_k \in W$. From the above example it should be clear that:

THEOREM 3.   *Let $F \in \epsilon_d(A)$ and W be a complete set of flags of F. For each $s \in S$ let $B(s)$ be the subsequence of $A(s)$ obtained by dropping all commands of the form $x \leftarrow f(y_1, \cdots, y_k)$, where $x \in W$. Then there is a flowchart F' which one can construct from F such that $F' \in \epsilon_d(B)$.*
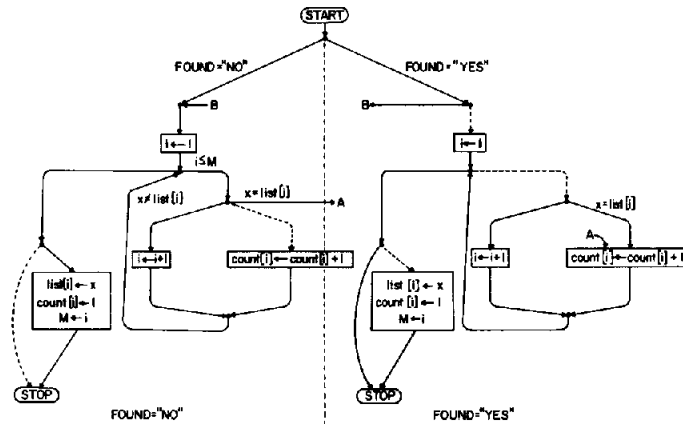

FIG. 3

5. Summary

The theorems of the previous sections are examples of results which might be misleading when applied to the problem of making an algorithm easy to understand. Theorem 1 means that D-charts are as powerful as flowcharts if we are allowed to add flags to a given flowchart. However, the form of the D-chart given in the proof of the theorem is clearly not a desirable expression of an algorithm. The additional flags in the D-chart merely represent the topology of the original flowchart, and this encoding of all the topology into flags does not necessarily make understanding the algorithm easy.

Theorem 3, on the other hand, shows that flags are superfluous since their effect can always be accounted for in the topology of a flowchart. This extreme is equally undesirable since a complex topology must be unraveled before an algorithm can be understood.

Finally, Theorem 2 indicates that we must necessarily permit the use of flags in D-charts if they are to be as powerful as arbitrary flowcharts. This does not mean, however, that D-charts are an inadequate means of expression.

REFERENCES

1. DIJKSTRA, E. Go to statement considered harmful. *Comm. ACM 11*, 3 (March 1968), 147–148.
2. BÖHM, C., AND JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9*, 5 (May 1966), 366–371.
3. KNUTH, D. E., AND FLOYD, R. W. Notes on avoiding "GOTO" statements. TR-CS 148, Comput. Sci. Dep., Stanford U., Stanford, Calif., Jan. 1970.