UNIVERSITY OF MARYLAND

COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

Technical Report TR-158 June 1971
GJ-1067 and NGL-21-002-008

On a Programming Language for Graph Algorithms

by

Werner C. Rheinboldt
Victor R. Basili
Charles K. Mesztenyi

This research was supported in part by the National Science Foundation and the National Aeronautics and Space Administration under Grant GJ-1067 and Grant NGL-21-002-008, respectively.

On a Programming Language for Graph Algorithms

by

Werner C. Rheinboldt, Victor R. Basili, and Charles K. Mesztenyi

Abstract

An algorithmic language, GRAAL, is presented for describing and implementing graph algorithms of the type primarily arising in applications. The language is based on a set algebraic model of graph theory which defines the graph structure in terms of morphisms between certain set algebraic structures over the node set and arc set. GRAAL is modular in the sense that the user specifies which of these mappings are available with any graph. This allows flexibility in the selection of the storage representation for different graph structures. In line with its set theoretic foundation, the language introduces sets as a basic data type and provides for the efficient execution of all set and graph operators. At present, GRAAL is defined as an extension of ALGOL 60 (Revised) and its formal description is given as a supplement to the syntactic and semantic definition of ALGOL. Several typical graph algorithms are written in GRAAL to illustrate various features of the language and to show its applicability.

On a Programming Language for Graph Algorithms[1)]

by

Werner C. Rheinboldt, Victor R. Basili, and Charles K. Mesztenyi[2)]

1. Introduction

During the past two decades, applications of graph theory have become increasingly important in a surprising number of fields. To a large extent, this has undoubtedly been caused by a growing trend toward computational approaches in these disciplines.

For the implementation of a graph theoretical algorithm on a computer, standard algorithmic languages, such as FORTRAN or ALGOL, are, in general, rather unsuitable. In fact, they are neither well-adapted to expressing basic graph theoretical operations, nor to describing and manipulating most of the data structures upon which these operations are defined. Although list processing languages provide for a more appropriate data structure, they tend to hide the graph theoretical nature of the algorithms besides leading to slow execution and large demands for storage. This points to the need for the development of special-purpose languages which facilitate the programming as well as the publication of graph algorithms.

[1)] This work was in part supported by Grant GJ-1067 from the National Science Foundation and Grant NGL-21-002-008 from the National Aeronautics and Space Administration.

[2)] All authors are with the Computer Science Center, University of Maryland, College Park, Maryland 20742.

In this article we propose such a language--named GRAAL (GRAph Algorithmic Language)--for use in the solution of graph problems of the type primarily arising in applications. These problems involve a wide variety of graphs of different types and complexity. This may include, for example, highly structured, directed or undirected graphs with multiple arcs and self loops and with various functions defined over the nodes and arcs, or very large, but sparse graphs in which only the adjacency relations between nodes are of interest and no further information is used. One of our objectives in the design of GRAAL was to allow for this wide range of possibilities with as little degradation as possible in the efficient implementation and execution of an algorithm designed for a specific type of problem. Our second objective relates to the earlier-mentioned need for a language which facilitates the design and communication of graph algorithms independent of the computer. In line with this we aimed at ensuring a concise and clear description of such algorithms in terms of data objects and operations natural to graph theory, that is, without introducing too many instructions required mainly by programming considerations.

In order to meet these apparently conflicting objectives, GRAAL was based on a strictly set algebraic model of graph theory which allows for considerable flexibility in the selection of the storage representation for different graph structures. The use of sets in graph algorithms is, of course, entirely natural, if only to express such concepts as the "set of all arcs incident with a node". However, in the development of a set theoretic data structure for graphs one soon faces complications

with ordered pairs of elements as they arise, for instance, in the usual definition of arcs as node pairs. In fact, either such pairs have to be treated as independent data objects, which requires separate instructions for them, or the Kuratowski definition $(x,y) = \{\{x\},\{x,y\}\}$ has to be employed, which leads to redundancies and the need for allowing sets of sets. Childs [1968a/b] has described a rather general approach to handling this problem. For the design of GRAAL we proceeded differently by using a model of graph theory which avoids the need for ordered pairs as well as for sets of sets. More specifically, the basic data objects of GRAAL are the elements of the power sets of the node set and arc set of a graph. Algebraic structures are imposed on these power sets and the basic graph operators defining the structure of the graph represent morphisms between these algebraic structures. GRAAL is a modular language in the sense that the user can specify which basic graph operators are available for any graph. This is the reason for the mentioned possibility of using various different storage representations for a graph structure in line with the specific nature of the problem at hand.

In view of its general set theoretic foundation, GRAAL incorporates sets as a new data type on the same level as integer, real, or Boolean variables. In order to allow for an effective implementation of the standard set operations, sets are assumed to contain only distinct elements which are ordered by an internal key. This key constitutes the unique internal identification for each basic element and each of these elements can in turn be used in any graph as either a node or an arc. In addition

to the data type "set" a data structure "list" has also been provided in GRAAL to allow stacking.

At present, GRAAL is defined as an extension of the revised ALGOL 60 language (Naur (ed.) [1963]). However, the language itself is relatively independent of ALGOL and could be redefined in terms of other algorithmic languages. In fact, a definition in terms of FORTRAN is now under way, in preparation for a first implementation of GRAAL in the form of a modified FORTRAN compiler. An ALGOL compiler version is planned for later.

During the past years, various graph algorithmic languages have been described in the literature. One of the earliest efforts along this line appears to have been a language of Tabory [1962] which was based on FORTRAN II and FLPL (FORTRAN-compiled List Processing Language). More recently, Friedman et al. [1969] (see also Friedman [1968]) developed an extension of LISP 1.5, called GRASPE 1.5, to allow graph processing on a list processing system. Another list-processing oriented language, HINT, has been described by Hart [1969]. The GTPL language of Read et al. [1969] (see also Read [1969]) is a system of FORTRAN II subroutines designed primarily for use in conjunction with graph theoretical studies such as the enumeration and cataloging of certain kinds of graphs and the analysis of some of their properties. The graph language ALLA of Wolfberg [1969], [1970] is a part of an interactive graphics system designed to allow the user to solve graph problems interactively with the aid of a display unit. As an aid in his work on approaches and techniques for analyzing the efficiency of graph algorithms, Chase [1970] developed a graph algorithmic

software package, GASP, consisting of a library of PL/1 procedures and of run-time macros. Last but not least, we mention Crespi-Reghizzi and Morpurgo [1968],[1970] who defined their graph language, GEA, as an extension of ALGOL 60.

Undoubtedly, there are other similar efforts not known to us. In particular, our list does not include languages which operate only on special types of graphs, such as the FORTRAN-based TREETRAN system of Pfaltz [1965] (revised [1970]) for the manipulation of rooted trees.

In Section 2 below we discuss the set theoretic foundation of GRAAL; then Section 3 presents the syntactic and semantic definition of the language as an extension of ALGOL 60; and finally, in Section 4 we give several examples of typical graph algorithms written in GRAAL.

2. Set Theoretic Foundations

Throughout this section, capital letters X,S,T, etc. stand for finite sets, and the basic set operations are indicated by the usual symbols "$\cup$" (union), "$\cap$" (intersection), and "$\sim$" (difference). In addition, we employ the symmetric sum

$$S \Delta T = (S \sim T) \cup (T \sim S) = (S \cup T) \sim (S \cap T),$$

which some authors also call the symmetric difference of S and T.

The cardinality of a set X is denoted by $|X|$, and $P(X)$ is the power set, that is, the set of all subsets of X. In addition, for $k = 0,1,\ldots,|X|$ we define

$$P_k(X) = \{S \in P(X) \mid |S| = k\}.$$

Thus, for instance, $P_0(X)$ contains only the empty set $\emptyset$, and, if $X = \{x_1,\ldots,x_n\}$, then the members of $P_1(X)$ are the n atomic sets $\{x_1\},\ldots,\{x_n\}$.

For any set X, it is, of course, well-known that $P(X)$ is a Boolean algebra under the operations union, intersection, and set complementation (in X). More specifically, $P(X)$ is a free Boolean algebra with the $|X|$ members of $P_1(X)$ as generators.

A different algebraic structure for $P(X)$ is obtained if we begin with the equally well-known observation that under the symmetric sum $P(X)$ is an Abelian group with $\emptyset$ as zero. Let $GF(2)$ be the binary Galois field consisting of the integers 0,1 under addition modulo two and standard multiplication. Then with the definition of the scalar product

$$\lambda S = \begin{cases} \emptyset \text{ if } \lambda = 0 \\ \\ S \text{ if } \lambda = 1 \end{cases}, \quad \forall S \in P(X),\ \lambda \in GF(2), \tag{2.1}$$

the Abelian group $P(X)$ becomes a vector space over $GF(2)$ in which now the $|X|$ members of $P_1(X)$ form a basis. It is sometimes useful to consider on this space the nondegenerate (but semidefinite) symmetric bilinear function

$$(S,T) = \begin{cases} 1 \text{ if } |S \cap T| \text{ is odd} \\ \\ 0 \text{ if } |S \cap T| \text{ is even} \end{cases}, \quad \forall S,T \in P(X). \tag{2.2}$$

Let X,Y be two sets. We denote by $B(X,Y)$ the class of all morphisms $\psi: P(X) \to P(Y)$ between the Boolean algebras $P(X)$ and $P(Y)$. The members of

$B(X,Y)$ shall be called Boolean mappings. Correspondingly, $L(X,Y)$ is defined as the class of all linear mappings $\psi: P(X) \to P(Y)$ between the vector spaces $P(X)$ and $P(Y)$.

It is well-known--and easily verified--that any Boolean mapping $\psi \in B(X,Y)$ can be written as

$$\psi S = \bigcup_{x \in S} \psi\{x\}, \quad \forall S \in P(X), \tag{2.3}$$

and, conversely, that when the image sets $\psi\{x\} \in P(Y)$ are given for all generators $\{x\} \in P_1(X)$ of $P(X)$, then (2.3) defines a member of $B(X,Y)$. A corresponding result holds, of course, for any linear mapping $\psi \in L(X,Y)$, that is, ψ is completely determined by the specification of $\psi\{x\} \in P(Y)$ for all basis elements $\{x\}$ in $P_1(X)$, and

$$\psi S = \bigcap_{x \in S} \psi\{x\}, \quad \forall S \in P(X). \tag{2.4}$$

Let G be a graph with node set V and arc set A. The elements of $P(V)$ and $P(A)$ then constitute the basic data objects for all operations on G under GRAAL. The structure of the graph is defined by certain Boolean or linear mappings between the two power sets, and different kinds of graphs are distinguished by the family of operators available for them. From a computational viewpoint there may be an additional distinction in terms of the storage representation used for the specific graph structure. In the remainder of this section, we define the basic graph operators presently included in GRAAL.

An undirected <u>pseudograph</u> is a triple $G = (V,A,\phi)$ consisting of a vertex (or node) set V, an arc set A, and an <u>incidence operator</u>

(2.5)
$$\text{(i)}\ \phi \in B(A,V)$$
$$\text{(ii)}\ \phi\{a\} \in P_1(V) \cup P_2(V), \quad \forall\, a \in A.$$

Thus, for any arc a, $\phi\{a\}$ is either the two-element subset of V consisting of the two distinct endpoints of a, or an atomic subset of V, in which case a is a self loop.

Following the terminology of Harary [1969] and others, we speak of a <u>multigraph</u> if in (2.5) the condition (ii) is replaced by (ii') $\phi\{a\} \in P_2(V)$, $\forall\, a \in A$. The unqualified term <u>graph</u> is used if, in addition, to (ii'), the restricted mapping $\phi: P_1(A) \to P_2(V)$ is one-to-one.

For any undirected pseudograph $G = (V,A,\phi)$ the <u>star operator</u> is defined as the Boolean mapping

(2.6) $$\sigma \in B(V,A),\ \sigma\{v\} = \{a \in A \mid v \in \phi\{a\}\},\ \forall\, v \in V.$$

Hence, for any node v, $\sigma\{v\}$ is the set of all arcs of G which are incident with v.

The star operator can be used to characterize the incidence structure of G. For this, note first that σ has the following properties:

(2.7)
$$\text{(i)}\ \sigma \in B(V,A).$$
$$\text{(ii)}\ P_1(A) \subset \bigcup_{v \in V} \sigma\{v\}$$
$$\text{(iii)}\ \sigma\{u\} \wedge \sigma\{v\} \wedge \sigma\{w\} = \emptyset \text{ for any three distinct elements } u,v,w \in V.$$

Let now V and A be any sets and σ any Boolean mapping which satisfies the three conditions of (2.7). Then (2.5) holds for the operator

$$(2.8) \qquad \phi \in B(A,V),\ \phi\{a\} = \{v \in V \mid a \in \sigma\{v\}\},\ \forall a \in A,$$

and evidently σ is the star operator of the resulting pseudograph $G = (V,A,\phi)$.

For the analysis of the topological structure of a pseudograph G the Boolean mappings ϕ and σ are not very convenient. The elements of $P(V)$ and $P(A)$ correspond in a natural way to the 0-chains and 1-chains of G, respectively, and accordingly, we can define the standard <u>boundary operator</u> ∂ and <u>coboundary operator</u> δ as the following linear mappings:

$$(2.9) \qquad \partial \in L(A,V),\ \partial\{a\} = \begin{cases} \phi\{a\} \text{ if } |\phi\{a\}| = 2 \\ \\ \emptyset \text{ otherwise} \end{cases},\ \forall a \in A,$$

$$(2.10) \qquad \delta \in L(V,A),\ \delta\{v\} = \{a \in \sigma\{v\} \mid |\phi\{a\}| = 2\},\ \forall v \in V.$$

Thus, ∂ maps each arc into the set of its two endpoints, provided they are distinct, and otherwise into the empty set. The set $\delta\{v\}$ consists of all arcs incident with v excluding any self loops.

Note that on a multigraph G we have $\partial\{a\} = \phi\{a\}$ for all $a \in A$ and $\delta\{v\} = \sigma\{v\}$ for all $v \in V$. Hence, the incidence structure of a multigraph can be defined in terms of the boundary operator or the coboundary operator.

It should be apparent that these definitions of the boundary and coboundary operators are equivalent with their usual definitions in terms of 0- and 1-chains. Accordingly, all the standard results about these operators are valid, such as the well-known formulae for the dimensions of their kernels and ranges. Moreover, under the bilinear function (2.2) we have

$$(2.11) \qquad (\partial S,T) = (S,\delta T), \quad \forall\, S \in P(A),\ T \in P(V),$$

that is, ∂ and δ are adjoint mappings.

If G is a graph, then each arc of G is uniquely determined by the two element set of its endpoints. This means in essence that the arcs are losing much of their own identity and hence that sometimes it may be expedient to work exclusively with the nodes. For this we introduce for a graph $G = (V,A,\phi)$ the <u>adjacency operator</u>

$$(2.12) \qquad \alpha \in B(V,V),$$

$$\alpha\{v\} = \{u \in V \mid \exists\, a \in \sigma\{v\}, \phi\{a\} = \{u,v\} \in P_2(V)\}.$$

Thus, α produces for each node v the set of all nodes u which form with v the (distinct) endpoints of some arc of G.

The adjacency operator again characterizes the incidence structure of the graph G. For this note that α has the properties

$$(2.13) \qquad \begin{aligned} &\text{(i)} \ \alpha \in B(V,V) \\ &\text{(ii)} \ v \notin \alpha\{v\}, \ \forall\, v \in V. \\ &\text{(iii)} \ u \in \alpha\{v\} \text{ if and only if } v \in \alpha\{u\}, \ \forall_{u,v} \in V. \end{aligned}$$

If, conversely, V is any set and α any Boolean mapping for which (2.13) holds, then the set

$$A = \{\{u,v\} \in P_2(V) \mid u \in \alpha\{v\}\}, \tag{2.14}$$

as well as the mapping

$$\phi \in B(A,V),\ \phi\{a\} = \{u,v\} \text{ if } a = \{u,v\},\ \forall a \in A, \tag{2.15}$$

are well-defined. Moreover, $G = (V,A,\phi)$ is a graph and α is the adjacency operator of G. We call the pair (V,α) the node form representation of the graph G.

The definitions of the various operators are easily carried over to directed graphs. A directed pseudograph shall be a quadruple $G = (V,A,\phi_+,\phi_-)$ consisting of a node set V, an arc set A, as well as a positive and negative incidence operator

$$\phi_+,\phi_- \in B(A,V),\ \phi_+\{a\},\ \phi_-\{a\} \in P_1(V),\ \forall a \in A. \tag{2.16}$$

In other words, $\phi_+\{a\}$ and $\phi_-\{a\}$ are atomic subsets of $P(V)$ consisting of the initial and terminal nodes of a, respectively. In many cases, it is convenient to use the combined incidence operator

$$\phi \in B(A,V),\ \phi\{a\} = \phi_+\{a\} \cup \phi_-\{a\},\ \forall a \in A. \tag{2.17}$$

As in the undirected case we speak of a directed multigraph if $\phi\{a\} \in P_2(V)$, for all $a \in A$, and of a directed graph (digraph) if, in addition, $\phi: P_1(A) \to P_2(V)$ is one-to-one.

The positive and negative star operators of the directed pseudograph G are defined by

$$\begin{aligned} &\sigma_+ \in B(V,A),\ \sigma_+\{v\} = \{a \in A \mid v = \phi_+\{a\}\},\ \forall\, v \in V, \\ &\sigma_- \in B(V,A),\ \sigma_-\{v\} = \{a \in A \mid v = \phi_-\{a\}\},\ \forall\, v \in V, \end{aligned} \tag{2.18}$$

and we introduce also the combined star operator

$$\sigma \in B(V,A),\ \sigma\{v\} = \sigma_+\{v\} \cup \sigma_-\{v\},\ \forall\, v \in V. \tag{2.19}$$

Thus, $\sigma_+\{v\}$ consists of all the arcs beginning in v and $\sigma_-\{v\}$ of those terminating in that node. Again, it follows immediately that σ_+ and σ_- can be used to characterize the incidence structure of a directed pseudograph. For this, conditions (i) and (ii) of (2.7) have to hold for both σ_+ and σ_- and (iii) can be replaced by (iii') $\sigma_+\{u\} \wedge \sigma_+\{v\} = \emptyset$, $\sigma_-\{u\} \wedge \sigma_-\{v\} = \emptyset$ for any $u \neq v$ in V.

The positive and negative boundary and coboundary operators of a directed pseudograph are now those <u>linear</u> mappings which coincide with the incidence and star operators on the appropriate family of atomic sets:

$$\begin{aligned} &\partial_+, \partial_- \in L(A,V),\ \partial_+\{a\} = \phi_+\{a\},\ \partial_-\{a\} = \phi_-\{a\},\ \forall\, a \in A, \\ &\delta_+, \delta_- \in L(V,A),\ \delta_+\{v\} = \sigma_+\{v\},\ \delta_-\{v\} = \delta_-\{v\},\ \forall\, v \in V. \end{aligned} \tag{2.20}$$

It is then natural to define the combined mappings

$$\begin{aligned} &\partial \in L(A,V),\ \partial\{a\} = \partial_+\{a\} \,\Delta\, \partial_-\{a\},\ \forall\, a \in A, \\ &\delta \in L(V,A),\ \delta\{v\} = \delta_+\{v\} \,\Delta\, \delta_-\{v\},\ \forall\, v \in V. \end{aligned} \tag{2.21}$$

Thus $\partial\{a\}$ is again the set of the endpoints of a if these endpoints are distinct, and the empty set, if they are not. Similarly, $\delta\{v\}$ is once more the set of all arcs incident with v excluding all self-loops. It is also readily seen that our definitions of these boundary and coboundary operators remain in agreement with the standard definitions, and hence that all the usual results are valid also in the directed case. This includes, in particular, the adjointness relation (2.11) for ∂ and δ which now also holds for the corresponding negative and positive operator pairs.

Finally, we define for a digraph $G = (V,A,\phi_+,\phi_-)$ the positive and negative adjacency operators by the relations

$$\alpha_+,\alpha_- \in B(V,V)$$

$$\alpha_+\{v\} = \{u \in V \mid \exists a \in \sigma_+\{v\},\ u = \phi_-\{a\}\},\ \forall v \in V \tag{2.22}$$

$$\alpha_-\{v\} = \{u \in V \mid \exists a \in \sigma_-\{v\},\ u = \phi_+\{a\}\},\ \forall v \in V.$$

Then the combined adjacency operator

$$\alpha \in B(V,V),\ \alpha\{v\} = \alpha_+\{v\} \cup \alpha_-\{v\},\ \forall v \in V \tag{2.23}$$

has again exactly the same meaning as in the undirected case. Moreover, α_+ and α_- may be used to characterize the incidence structure of a digraph. Here conditions (i) and (ii) of (2.13) have to hold for both α_+ and α_- and (iii) is replaced by (iii') $u \in \alpha_+\{v\}$ if and only if $v \in \alpha_-\{u\}$, $\forall u,v \in V$. This defines the node form representation (V,α_+,α_-) of a digraph.

3. Syntax and Semantics of GRAAL

GRAAL is defined as an extension of ALGOL 60. The formal description presented here is simply a supplement to the syntactic and semantic definition in the Revised ALGOL Report (Naur (ed.) [1963]). Each of the following subsections begins with some BNF grammar rules of ALGOL which are extended in GRAAL. The extensions are the metalinguistic symbols appearing after the double slash (||). The rest of the subsection then contains the syntactic definition of these new metalinguistic variables along with some examples and a verbal explanation of their semantics. The grammatic rules of ALGOL unaffected by the definition of GRAAL are not repeated here.

A. Declarations

Syntax

<type> ::= real | integer | Boolean || set | alpha

<declaration> ::= <type declaration> | <array declaration>|
<switch declaration> | <procedure declaration>||
<graph declaration> | <list declaration>|
<property declaration>

<graph declaration> ::= graph <graph list>

<graph list> ::= <graph specification> | <graph list>,<graph specification>

<graph specification> ::= <graph identifier>[<integer>] |
<graph identifier>[<string>]

<graph identifier> ::= <identifier>

<list declaration> ::= list <list list>|

<local or own type> list <list list>

<list list> ::= <list identifier> | <list list>,<list identifier>

<list identifier> ::= <identifier>

<property declaration> ::= property <property list>|

<local or own type> property <property list>

<property list> ::= <property identifier>|

<property list>,<property identifier>

<property identifier> ::= <identifier>

Examples

```
set A,B,C;
graph G[1],H['directed pseudograph'];
real list a;
set list SL;
real property capacity;
set property L;
```

Semantics

Two new data types are introduced. The alpha variable represents the normal alphanumeric variable already available in most implementations of algebraic languages. In GRAAL, sets constitute a new basic data type rather than a data structure. An atomic set is a set consisting of one item, and any set is either empty or a union of atomic sets; (see subsection C below).

There are three new data structures, namely, graphs, lists, and properties. Graphs represent specific data structures together with certain operations for manipulating them. The graph declaration identifies the type of data structure used and the family of graph operators available with it. The language is modular in the sense that, in general, only some of the possible graph operators are usable with any specific graph. Four modules are presently defined in the language; they are identified in subsection C below.

A list is a doubly-open, linked list structure which may be used as a stack or a queue. Its order is established by the sequence in which the user links the values of the variables of the declared type. It offers a locally dynamic alternative to the array for storing variables.

A property may be associated with any atomic set. The property declaration establishes the type of the property. When no type declarator is given, the real type is understood. The property for a particular atomic set exists and may be referenced only after it has been assigned a value (i.e., storage for a property of an atomic set is dynamically allocated). If a property is referenced which does not exist for the specified atomic set, then a default value is returned. For alpha properties this will be blank, for set properties empty, for Boolean properties false, and for real as well as integer properties, the largest available machine number. The standard function check included in subsection H below uses these default conditions to test for the existence of a property for a given atomic set.

B. Variables

Syntax

<variable> ::= <simple variable> | <subscripted variable>‖
<property variable>

<property variable> ::= <property identifier>.(<variable>)

Examples

capacity.(x) := 2.3

m := label.(y)

Semantics

The new property variable resembles the subscripted variable in that it requires an argument. This argument is enclosed between the "dotted" left parenthesis '.('and the right parenthesis')'; it must be an atomic set. If a nonatomic set is referenced in the argument, the first element of that set is taken as the default argument (see subsection C below for the ordering of sets). The value of the property variable is not defined if any argument other than a set variable is specified.

C. Assignment Statement

Syntax

<assignment statement> ::= <left part list><arithmetic expression>|
<left part list><Boolean expression>‖
<left part list><set expression>|
<left part list><list expression>|
<left part list> empty

<expression> ::= <arithmetic expression> | <Boolean expression>|
<designational expression> || <set expression>|
<list expression>

<set expression> ::= <set union> | <set expression> ~ <set union>

<set union> ::= <set sum> | <set union> ∪ <set sum>

<set sum> ::= <set intersection> | <set sum> Δ <set intersection>

<set intersection> ::= <set primary> | <set intersection> ∩ <set primary>

<set primary> ::= <variable> | (<set expression>) | <function designator>|
<subset operator designator> | <graph operator designator>|
<atomic set operator designator>

<subset operator designator> ::= subset (<simple variable>,
<Boolean expression>)|
subset (<simple variable> in <set expression>,
<Boolean expression>)

<atomic set operator designator> ::= create|
create (<atom definition list>)|
atom (<simple arithmetic expression>)

<atom definition list> ::= <atom definition>|
<atom definition list>,<atom definition>

<atom definition> ::= <property identifier>:<variable>

<graph operator designator> ::= <graph structure designator>|
<basic graph operator designator>

<graph structure designator> ::= <structure operator>(<set expression>,
<graph identifier>)

<structure operator> ::= <mod 1 structure operator>|
<mod 2 structure operator>|
<mod 3 structure operator>|
<mod 4 structure operator>

<mod 4 structure operator> ::= adj

<mod 3 structure operator> ::= padj | nadj|
<mod 4 structure operator>

<mod 2 structure operator> ::= inc | star | bd | cob

<mod 1 structure operator> ::= pinc | ninc | pstar | nstar|
pbd | nbd | pcob | ncob|
<mod 2 structure operator>

<basic graph operator designator> ::=
<basic graph operator>(<graph identifier>)

<basic graph operator> ::= <mod 1 basic graph operator>|
<mod 2 basic graph operator>|
<mod 3 basic graph operator>|
<mod 4 basic graph operator>

<mod 1 basic graph operator> ::= <mod 2 basic graph operator>

::= arcs | nodes

<mod 3 basic graph operator> ::= <mod 4 basic graph operator>

::= nodes

<list expression> ::= <list element> | <list expression> o <list element>

<list element> ::= <number> | <variable> | <list identifier>|

(<expression>) | <function designator> | <list operator designator>

<list operator designator> ::= <list operator> (<list expression>)

<list operator> ::= f | fd | ℓ | ℓd

Examples

S := X ∪ Y ∩ C ~ D Δ M

L := bd(X,G) ∩ cob(Y,G)

M := nodes (G) ∪ arcs (G)

S := subset (x in star (Y,G), capacity.(x) > 0)

x := create (name: i, capacity: k)

S := S ∪ create

X := atom (1) ∪ atom (2) ∪ atom (i+1)

S := subset (x, cap.(x) > 0 ∨ cap.(x) < 20)

L := L o a o 3 o (a+b) o cap.(x)

L := L o K o M

Semantics

A set expression is a rule for creating, referencing, and manipulating sets. Each atomic set carries a sequence number which is assigned to it at the time of its creation. A set is a union of atomic sets

ordered in ascending order of their sequence number. This ordering allows for an efficient manipulation of sets. All sequence numbers assigned to atomic sets are retained in an element sequence. This is an ordered internal structure serving the dual purpose of cataloging the atomic sets which have been created so far and of providing the linkage between an atomic set and the properties which are assigned to it. It is envisioned that the ith location of the element sequence is the start of the list of property-value pairs associated with the ith atomic set. A property-value pair is added to the list when a value is assigned to a property for an atomic set at execution time.

The create operator may or may not include an argument. If given, the argument is a list of pairs each consisting of a property and of a variable designating a value for it. The element sequence is searched for an atomic set for which all the named properties exist and are presently assigned the specified values. If a (complete) match is found the create operator returns the corresponding atomic set. If no match (or only a partial match) occurs, a new element with the next sequence number and with the stated property values is added to the element sequence and an atomic set carrying this sequence number is created and returned. If the create operator carries no argument, only the last action occurs, that is, a new element with the next sequence number is added to the element sequence and an atomic set with this new number is returned.

The atom operator returns the atomic set whose sequence number is given by the arithmetic expression in its argument. If no atomic set

with this number exists or if the expression is not integer-valued, the empty set is returned.

The subset operator constructs a set consisting of atomic sets which satisfy the specified Boolean expression. Depending on the form of the argument of the subset operator, either all atomic sets cataloged in the element sequence are tested or only those contained in the set specified by the given set expression.

As stated earlier, different kinds (modules) of graphs are distinguished by the type of the data structure used to represent them and by the family of graph operators provided with this structure. The present four graph modules are distinguished only in terms of their graph operators. Additional modules which may or may not duplicate one of the family of operators, but which refer to different data structures, will be added in the implementation. The present four modules are mod 1 'directed pseudograph', mod 2 'undirected pseudograph', mod 3 'directed graph in node form', and mod 4 'undirected graph in node form'.

The graph operators construct sets on the basis of a given graph structure. The basic graph operators nodes or arcs return the set consisting of all atomic sets that were assigned either as nodes or as arcs to a specified graph. The structure operators require as an argument a set expression which designates either a set of nodes or of arcs of the specified graph. The various possible operators were formally defined in Section 2; those presently included in the language are the incidence operator inc, the positive and negative incidence operators pinc and ninc, the star operator star, the positive and negative star operators pstar and nstar, the boundary operator bd, the positive and

negative boundary operator pbd and nbd, the coboundary operator cob, the positive and negative coboundary operator pcob and ncob, the adjacency operator adj, and the positive and negative adjacency operator padj and nadj. If for any of these operators an argument set is specified which contains an atomic set not belonging to the required node or arc set of the graph, the empty set is returned as a default value.

The binary set operators have the standard set theoretic meaning. In increasing precedence order they are difference (~), union ($\cup$), symmetric sum (Δ), and intersection ($\cap$). The ordering of sets makes the execution of these operations fairly efficient.

The semantic interpretation of the ALGOL assignment statements remains valid for the extended definition of these statements in GRAAL. In particular, the type associated with all variables and procedure identifiers of a left part list must be the same. Moreover, if the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are to be invoked. The specific form of the various new transfer functions is left to the implementation. A reasonable possibility for transfers between set type and real/integer type might be as follows: If x is a set variable and y an integer or real variable, then the statement x := y is equivalent with x := atom (entier(y)), while y := x is equivalent with y := count (x) where count is a standard function defined in subsection H. As in most algebraic languages, including ALGOL, the

copy rule applies to an assignment, i.e., in the simple set assignment statement, S := T, a copy of T is assigned to S. Thus, each set corresponds to a unique set variable.

As stated earlier, a list structure is basically a stack or a queue. To build the list, items are concatenated together. When a list of n items is concatenated with a list of m items, the resulting list contains n+m items. To remove items from a list, there are four operators: f returns the first item of a list, while fd yields this first item and deletes it from the list; similarly ℓ returns the last item of a list, and ℓd gives the last item and deletes it from the list. A list must be declared as to type; if no declarator is given, the real type is understood. A list operates similar to an array in that a copy of each item is stored in it.

D. Unlabeled Basic Statement

Syntax

<unlabeled basic statements> ::= <assignment statement> |
<go to statement> | <dummy statement> |
<procedure statement> || <link statement>

<link statement> ::= assign (<graph identifier>,<link form>)
detach (<graph identifier>,<set expression>) |
detach (<graph identifier>)

<link form> ::= <isolated node form> | <node graph form> |
<full graph form>

<isolated node form> ::= <node>

<node graph form> ::= <node><directional><node>

<full graph form> ::= <node><directional><node> to <arc>

<directional> ::= → | -

<node> ::= <arc> ::= <variable> | <atomic set designator>

Examples

assign (G,n1 → n2 to a1);

assign (G, n - m);

assign (G,n);

detach (H,S ∪T);

detach (G);

Semantics

The assign operator constructs the incidence structure of a graph. This graph structure may be represented by defining some appropriately chosen family of the graph structure operators as a set property of each node or arc in that graph. The full link form contains three atomic sets and specifies that in the given graph the last one of these is to be an arc with the first two as its endpoints. The connection may be specified as directed, using '→', or as undirected, using '-'. Isolated nodes may be inserted by listing only one atomic set. If the graph was declared to be in node form, the arc specification is deleted, and when included it is ignored. The assign statement is not executed if the specified assign action was taken earlier. If an arc is specified which was assigned earlier to different endpoints the earlier assign action is superseded by the later one. The assign statement is treated as a dummy statement if any of the sets contained in the link form is empty or nonatomic, or if the specified assign action is not permitted

for the particular graph module.

The detach statement allows the deletion of specified parts from the incidence structure of a graph. In sequence, for each atomic set contained in the specified set one of the following steps are taken: If the element is an arc of the given graph, this arc is removed; if it is a node, this node is removed as well as all arcs incident with it; and if neither case applies, the element is bypassed. If no set is specified, then all nodes and arcs are removed from the graph.

E. Statement

Syntax

<statement> ::= <unconditional statement> | <conditional statement>|
<for statement> || <for all statement> | <while statement>|
<removal statement>

<conditional statement> ::= <if statement>|
<if statement> else <statement> | <if clause><for statement>
<label>:<conditional statement>||
<if clause><for all statements>

<for all statement> ::= <for all clause><statement>|
<label> : <for all statement>

<for all clause> ::= for all <for all element> do

<for all element> ::= <set for all element> | <list for all element>

<set for all element> ::= <variable> in <set expression>

<list for all element> ::= <variable> in <list expression >

<while statement> ::= <while clause><statement>|
<label> : <while statement>

<while clause> ::= while <Boolean expression> do

<removal statement> ::= delete (<set expression>) |

erase (<property identifier>,<set expression>) |

erase (<property identifier>)

Examples

for all x in X∩Y do if capacity.(x) > 0 then M := M∪x;

for all i in List do n := n+1;

while ¬(S⊆T) do T := S;

delete (nodes (G)∪ arcs (G));

erase (capacity, S);

erase (length);

Semantics

The for all clause causes the statement S which follows it to be executed zero or more times, once for each element in the specified set or list. The dummy variable in the for all clause takes on as its value the value of every element in the set or list, one at a time in sequence. The while clause causes the statement S which follows it to be executed zero or more times, as long as the value of the Boolean expression is true. Control passes to the next statement when the value of the Boolean expression is false. The erase statement removes the specified property-value pair from all members of the given set. If no set is specified, the property is removed from all the atomic sets for which it exists. The delete statement removes all atomic sets in the designated set as well as their associated properties from the catalog in the element sequence. If a removed atomic set is referenced, an error condition occurs.

F. Boolean Primary

Syntax

<Boolean primary> ::= <logical value> | <variable>|

<function designator> | <relation>|

(<Boolean expression>) ‖ <set relation>

<set relation> ::= <extended set expression>

<set relational operator><extended set expression>

<extended set expression> ::= <set expression> | empty

<set relational operator> ::= = | ≠ | ⊆ | ⊇

Examples

X ⊆ Y

X = Y

Y ≠ empty

capacity.(x) = 2

Semantics

The metalinguistic variable <Boolean primary> has been extended to include relations among sets. The set relational operators equal (=), not equal (≠), contained in (⊆ or ⊇). In this connection the set expression has been extended to include empty in order to check if a set is empty.

G. Procedures

Syntax

<specifier> ::= string | <type> | array | <type> array |
label | switch | procedure | <type> procedure |
list | <type> list | property | <type> property |
graph

<actual parameter> ::= <string> | <expression> | <array identifier> |
<switch identifier> | <procedure identifier> |
<property identifier> | <graph identifier>

Examples

```
procedure test (G, capacity, List);
    real list List; graph G; integer property capacity;
set procedure S(G,T,A);
    graph G; set T; array A;
test (Graph, Property, List)
X := S(Graph, Set, Array) ∪ M
```

Semantics

The specifiers required in procedure and function declarations, as well as the actual parameters needed for the corresponding statements, have been extended in a normal way to include the new data types and structures.

H. Standard Functions

Add to the list of standard functions:

size (T)	for the number of elements in the set of list T
parity (T)	which is true if size (T) is odd, else false, where T is a set or list
index (x,T)	which returns the index of the place taken by the element x in the set or list T
elt (i,T)	which returns the ith element of the set or list T
check (f(S))	which is true if the property variable f is defined on the set S and false otherwise
count (S)	which returns the sequence number of a given atomic set S. If S is not an atomic set, it returns the sequence number of its first element
maxcount	returns the sequence number of the last atomic set created

Each of these standard functions is programmable as a procedure in GRAAL.

4. Examples of GRAAL Programs

In this section we present several typical graph algorithms in the form of GRAAL procedures. The principal aim here is to illustrate some of the main features of GRAAL as they may be used in practice; accordingly, no particular attempt was made to optimize the algorithms or even to include all possible error checks.

GRAAL does not require any specific format for the input of a graph. In fact, once typical input/output instructions have been added to ALGOL, any of the standard methods of representing graphs may be used to read in the structure. We give here only two simple examples.

```
procedure readone (G);
graph G;
comment The procedure assumes that the first record provides
the sizes  n  and  m  of the node and arc set and that then
m  records are supplied each containing three integers.  Any
such triple (k,i,j) satisfies 1 ⩽ k ⩽ m, 1 ⩽ i,j ⩽ n and
signifies that the kth arc has the ith node as initial, and
the jth node as terminal vertex;
begin integer n,m,k,i,j,ℓ; set x;
     read (n,m);
     for ℓ=1 step 1 until n+m do x := create;
     for ℓ=1 step 1 until m do
     begin read (k,i,j);
          assign (G, atom(i) → atom(j) to atom(n+k))
     end
end
```

Note that the default feature for the assign instruction provides some check for the validity of the input triples. A more extensive check may, of course, be desirable.

```
procedure readtwo (G, name);
graph G; alpha property name;
comment A read-in procedure 'read (buffer)' is assumed to be
available which allows the input of a variable-length record
of alphanumeric words into the alpha list buffer. The undirected
graph G is represented in node form and is read-in in terms of
paths, that is, as sequences of nodes forming paths in G. The
input is terminated with a record containing the single word
'last';
begin alpha list buffer; set x,y;
     read (buffer);
     while f(buffer) ≠ 'last' do
     begin x := create (name: fd(buffer));
          if buffer = empty
               then assign (G,x)
               else while buffer ≠ empty do
                    begin y := create (name: fd(buffer));
                         assign (G,x - y);
                         x := y
                    end;
          read (buffer)
     end
end
```

The next three examples concern the derivation of some simple new graphs from an existing graph G. In all cases, G is assumed to be an undirected pseudograph; for directed graphs only the assign instructions need to be changed.

```
procedure subgraph (G, N, SubG);
graph G, SubG; set N;
comment The procedure sets up the subgraph of G which has a given
set N of nodes of G as node set;
```

```
begin set S,x,y,a;
     while N ≠ empty do
     begin x := elt(1,N);
          S := subset (a in star (x,G), inc (a,G) ⊆ N);
          N := N ~ x;
          if S = empty then assign (Subg,x)
               else for all  a  in  S  do
                    begin y := inc (a,G) ~ x;
                         if y = empty then y := x;
                         assign (Subg, x - y to  a)
                    end
     end
end

procedure linegraph (G, LineG);
graph G, LineG;
comment This procedure sets up the line graph of G, that is, the
graph which has the arcs of G as nodes and in which two nodes are
adjacent whenever the corresponding arcs of G are:
begin set S,R,x,a,b;
     for all  x  in nodes (G) do
     begin S := R := star (x,G);
          for all  a  in  S  do
          begin if x = inc (a,G) then assign (LineG, a-a to create);
               R := R ~ a;
               for all  b  in  R  do assign (LineG, a-b to create)
          end
     end
end

procedure condense (G,L, ConG,ref);
graph G, ConG; set list L; set property ref;
comment The list  L  is assumed to contain a family of sets represent-
ing a partition of the node set of  G.  The procedure sets up a condensed
graph which has the members of  L  as nodes and in which two nodes are
adjacent if there is at least one arc between the corresponding sets
of nodes in  G.  The property 'ref' of the nodes of ConG remembers the
sets of  L;
```

```
begin set S,T,x;
     while L ≠ empty do
     begin S := fd(L);
          x := create (ref: S);
          assign (ConG,x);
          for all  T  in  L  do
          if inc (star(S,G),G)∧T ≠ empty then
               assign (ConG, x - create (ref: T) to create)
     end
end
```

The following four algorithms relate to the analysis of the topological structure of a pseudograph. They apply equally well if the graph is directed or undirected.

```
procedure cocycles (G,C);
graph G; set list C;
comment This procedure determines a basis for the cocyle space
by finding the node sets of all connected components of  G;

begin set N,A,S,T;
     N := nodes (G);
     while N ≠ empty do
     begin A := T := empty;
          S := elt(1,N);
          while S ≠ empty do
          begin T := T∪S;
               A := star(S,G) ~ A;
               S := inc(A,G) ~ T
          end
          C := C ∘ T;
          N := N ~ T
     end
end
```

```
procedure spantree (G,u,Tree);
graph G, Tree; set u;
comment This procedure generates a directed spanning tree with
root  u  for the connected component of  G  containing the node  u;
begin set S,T,w,x,y,a;
     assign (Tree,u);
     S := u;
     T := cob(u,G);
     while T ≠ empty do
     begin for all  a  in  T  do
          begin w := bd(a,G);
                y := w ~ S;
                if y ≠ empty then begin S := S∪y;
                                        x := w ~ y;
                                        assign (Tree, x → y to  a)
                                  end
          end;
          T := cob(S,G)
     end
end


procedure fundcycles (G,Tree,Cycles);
graph G, Tree; set list Cycles;
comment 'Tree' is assumed to be a directed spanning tree of one of
the components of  G.  From this spanning tree this procedure
generates, in a standard manner, a basis for the cycle space of
the particular component;
begin set X,S,T,a;
     X := star (nodes(Tree),G) ~ arcs (Tree);
     for all  a  in  X  do
     begin S := a;
          T := inc(a,G);
          if size (T) ≠ 1 then
```

```
            while T ≠ empty do
            begin T := ncob(T,Tree);
                  if T ≠ empty then
                  begin S := S Δ T;
                        T := pbd(T,Tree)
                  end
            end;
            Cycles := Cycles ∘ S
      end
end
```

```
procedure fundcut (G,Tree,Cuts);
graph G,Tree; set list Cuts;
comment Again 'Tree' is assumed to be a directed spanning tree
of a component of G, and from 'Tree' this procedure generates
in the standard manner a basis of the coboundary space of the
component;
begin set a,S,T;
      for all a in arcs (Tree) do
      begin S := empty;
            T := a;
            while T ≠ empty do
            begin S := S∪nbd(T,Tree);
                  T := pcob(S,Tree) ~ T
            end;
         ℓ: Cuts := Cuts ∘ cob(S,G);
      end
end
```

Note that instead of the statement ℓ, it might be more efficient to store in 'Cuts' merely the node sets S and to generate the actual cut sets cob (S,G) only when needed.

We end this section with a larger program to show the interplay between different features of GRAAL. For this we chose a shortest-path algorithm given by Pohl [1969] involving a bidirectional search.

```
procedure shortpath (G,start,term,length,inf,m,path);
graph G; set start, term; real property length; real inf, m;
    set list path;
comment G is a digraph in which each arc has a given nonnegative
length. The procedure finds a shortest path from node 'start' to
node 'term' and returns it in the list 'path'. If no such path
exists, the list will be empty. The real number 'inf' represents
infinity, it is assumed to be larger than the sum of the length
of all arcs of G. The length of the final path will be in m,
and this number will be equal to inf, if no path exists;
begin set S,SR,T,TR,w,x,y,z,u;
    real property sdist, tdist;
    set property in, out;
    boolean flag; real a,b, smin, tmin;

    comment The notation is as follows:
    S (or T) set of nodes reached from 'start' (or 'term')
    SR (or TR) nodes not in S (or T) but reachable therefrom
        along one arc
    sdist.(x) (or tdist.(x)) current distance between 'start'
        (or 'term') and x
    in.(x) (or out.(x)) current arc leading to (or from) x
    smin (or tmin) minimal distance from 'start' (or 'term')
        to SR (or TR);

    comment Initialization;
    sdist.(start) := tdist.(term) := 0;
    S := SR := start;
    T := TR := term;
    smin := tmin := 0;
    flag := false;
    m := 0;
```

```
     comment Insert a fictitious arc  w  from 'start' to 'term'
     with length inf.  This ensures that there is at least one
     path between these two nodes;
     w := create;
     length.(w) := inf;
     assign (G, start → term to w);

     comment Test for completion and decision to proceed either
     from 'start' or 'term';
decide: if m := inf then go to nopath;
     if flag then go to found;
     if smin ≤ tmin then go to fromstart else go to fromterm;
     comment Proceed from start and find minimal distance in SR;
fromstart: m := inf;
     path := empty;
     for all  x  in  SR  do
     begin if check (sdist.(x)) then a := sdist.(x) else a := inf;
          if a < m then begin m := a; path := x end
                   else if a = m then path : = x ∘ path
     end;
     smin := m;

     comment Transfer set memberships and determine current
     distances and in arcs;
     for all  x  in path do
     begin if (¬flag) ∧ (x ⊆ T)  then begin flag := true; u := x
                                         end;
          SR := SR ~ x;
          S := S ∪ x;
          for all  z  in pcob (x,G) do
          begin y := nbd (z,g);
                b := m + length.(z);
                if check (sdist.(y)) then a := sdist.(y) else a := inf;
```

```
                if a > b then begin sdist.(y) := b;
                                    in.(y) := z;
                                    SR := SR ∪ y
                              end
            end
        end;
        go to decide;
        comment Proceed from 'term';
fromterm: m := inf;
        path := empty;
        for all x in TR do
        begin if check (tdist.(x)) then a := tdist.(x) else a := inf;
            if a < m then begin m := a; path := x end
                else if a = m then path := x ∘ path
        end;
        tmin := m;
        for all x in path do
        begin if (¬flag) ∧ (x ⊆ S) then begin flag := true; u := x end;
            TR := TR ~ x;
            T := T ∪ x;
            for all z in ncob (x,G) do
            begin y := pbd (z,G);
                b := m + length.(z);
                if check (tdist.(y)) then a := tdist.(y) else a := inf;
                if a > b then begin tdist.(y) := b;
                                    out.(y) := z;
                                    TR := TR ∪ y
                              end
            end
        end;
        go to decide;
```

```
nopath: path := empty;
     go to exit;
     comment Breakthrough, check for other nodes which have
     been reached from both sides, then establish a shortest
     path;
 found: m := sdist.(u) + tdist.(u);
     y := u;
     for all  x  in T ∩ (S ∪ SR) do
     begin a := sdist.(x) + tdist.(x);
           if a < m then begin m := a; y := x end
     end;
     u := y;
     path := u;
     x := u;
     while x ≠ start do
          begin z := in.(x);
                y := pbd(z,G);
                path := y ∘ z ∘ path;
                x := y
          end;
     x := u;
     while x ≠ term do
          begin z := out.(x);
                y := nbd(z,G);
                path := path ∘ z ∘ y;
                x := y
          end;
exit: remove (w)
end
```

5. References

Chase, S. [1970]. Analysis of algorithms for finding all spanning trees of a graph, Department of Computer Science Report 401, Univ. of Illinois, Urbana, Illinois.

Childs, D. [1968a]. Feasibility of a set-theoretic data structure -- a general structure based on a reconstituted definition of a relation, Proc. IFIP Congress 68, I62-I72.

Childs, D. [1968b]. Description of a set-theoretic data structure, Proc. Fall Joint Computer Conference 68, 557-564.

Crespi-Reghizzi, S., and Morpurgo, R. [1968]. A graph theory oriented extension of ALGOL, Calcolo 5, 1-43.

Crespi-Reghizzi, S., and Morpurgo, R. [1970]. A language for treating graphs, Comm. ACM 13, 319-323.

Friedman, D. [1968]. GRASPE graph processing: a LISP extension, Computation Center Report TNN-84, Univ. of Texas, Austin, Texas.

Friedman, D., Dickson, D., Fraser, J., and Pratt, T. [1969]. GRASPE 1.5, a graph processor and its application, Department of Computer Science Report RS1-69, Univ. of Houston, Houston, Texas.

Harary, F. [1969]. "Graph Theory", Addison-Wesley, Reading, Massachusetts.

Hart, R. [1969], HINT: a graph processing language, Institute for Social Science Research Technical Report, Michigan State Univ., East Lansing, Michigan.

Naur, P. (ed.) [1963]. Revised report on the algorithmic language ALGOL 60, Comm. ACM 6, 1-17.

Nievergelt, J. [1970]. Software for graph processing, SIGSAM Bulletin No. 14.

Pfaltz, J. [1965] (revised [1970]). TREETRAN - A FORTRAN IV subroutine package for manipulation of rooted trees, Computer Science Center Technical Report 65-23, Univ. of Maryland, College Park, Maryland.

Pohl, I. [1969]. Bi-directional and heuristic search in path problems, Computer Science Department Technical Report CS-136, Stanford Univ., Stanford, California.

Read, R., King, C., Cadogan, C., and Morris, P. [1969]. The application of digital computer techniques to the study of graph-theoretical and related combinatorial problems, Computer Centre Report on Project 1026-66, Univ. of the West Indies, Jamaica.

Read, R. [1969]. Teaching graph theory to a computer, in "Recent Progress in Combinatorics", Academic Press, New York, New York.

Tabory, R. [1962]. Premiers elements d'un language de programmation pour le traitement en ordinateur des graphes, in "Symbolic Languages for Data Processing", Gordon and Breach, New York, New York.

Wolfberg, M. [1969]. An interactive graph theory system, Moore School of Electrical Engineering Report 69-25, Univ. of Pennsylvania, Philadelphia, Pennsylvania.

Wolfberg, M. [1970]. An interactive graph theory system, Technical Report CA-7003-0211, Massachusetts Computer Associates, Wakefield, Massachusetts.