BIT 12 (1972), 220–241

ON A PROGRAMMING LANGUAGE FOR GRAPH ALGORITHMS[1]

WERNER C. RHEINBOLDT, VICTOR R. BASILI
and CHARLES K. MESZTENYI

Abstract.

An algorithmic language, GRAAL, is defined, as an extension of ALGOL 60 (Revised), for describing and implementing graph algorithms of the type arising in applications. It is based on a set algebraic model of graph theory which defines the graph structure in terms of user specified morphisms between certain set algebraic structures over the node and arc set. Several examples of graph algorithms written in GRAAL are included.

1. Introduction.

For the implementation of a graph 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 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.

In this article we propose such a language—named GRAAL (*GRA*ph *A*lgorithmic *L*anguage)—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; and one of the objectives in the design of GRAAL was to allow for this 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 need for a language which facilitates the design and communication of graph algorithms independent of the

Received December 13, 1971. Revised February 8, 1972.

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

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 objectives, GRAAL was based on a 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 oriented 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. Childs [2], [3] has described a rather general approach to set theoretic data structures. For the design of GRAAL we proceeded more simply by using a model of graph theory in which the basic data objects are the elements of the power sets of the node and arc set. 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 provides for the 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 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 [10]. However, the language itself is relatively independent of ALGOL and can be redefined in terms of other algorithmic languages; in fact, a FORTRAN-based version is presently being developed.

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 [17] which was based on FORTRAN II and FLPL (FORTRAN-compiled List Processing Language). More recently, Friedman et al. [7] (see also Friedman [6] as well

as Pratt and Friedman [14]) 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 [9]. The GTPL language of Read et al. [15] (see also Read [16]) is a system of FORTRAN II subroutines designed primarily for use in conjunction with theoretical studies of graphs. The graph language ALLA of Wolfberg [18], [19] is a part of an interactive graphics system and allows the solution of graph problems with the aid of a display unit. As an aid in his work on the efficiency of graph algorithms, Chase [1] developed a graph algorithmic software package, GASP, consisting of a library of PL/1 procedures and run-time macros. Finally, we mention Crespi-Reghizzi and Morpurgo [4], [5] who defined their graph language, GEA, as an extension of ALGOL 60. Undoubtedly, there are other similar efforts not known to us. Also, our list does not include languages which operate only on special types of graphs, as, for instance, the TREETRAN system of Pfaltz [12] for the manipulation of rooted trees.

2. Set theoretic foundations.

In 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), "$\sim$" (difference), and "$\triangle$" (symmetric sum). For any set X, the cardinality is denoted by $|X|$, $P(X)$ is the power set, and we define

$$P_k(X) = \{S \in P(X) \mid |S| = k\}, \quad k = 0, 1, \ldots, |X| . \tag{2.1}$$

In particular, $P_0(X)$ 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 $\{x_1\}, \ldots, \{x_n\}$.

It is well-known that under union, intersection, and complementation (in X), $P(X)$ becomes a Boolean algebra with the members of $P_1(X)$ as generators. To obtain another algebraic structure, let $GF(2)$ be the binary Galois field with the integers 0, 1 as elements. Then $P(X)$ becomes a vector space over $GF(2)$ if the symmetric sum is used as addition and the scalar product is defined by $\lambda S = \emptyset$ for $\lambda = 0$ and $\lambda S = S$ for $\lambda = 1$. The elements of $P_1(X)$ now form a basis.

For any sets X, Y we denote by $B(X, Y)$ the class of all morphisms $\psi: P(X) \to P(Y)$ between the Boolean algebras $P(X), P(Y)$. Any $\psi \in B(X, Y)$ is uniquely characterized by the image sets $\psi\{x\} \in P(Y)$ of the generators $\{x\} \in P_1(X)$ and

$$\psi S = \bigcup_{x \in S} \psi\{x\}, \quad \forall\, S \in P(X) . \tag{2.2}$$

Correspondingly, we define $L(X, Y)$ as the class of all linear mappings $\psi\colon P(X) \to P(Y)$ between the vector spaces $P(X), P(Y)$. Then, instead of (2.2), we have for any $\psi \in L(X, Y)$ the representation

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

Let G be a graph with node set V and arc set A. The elements of $P(V)$ and $P(A)$ constitute the basic data objects for all operations on G under GRAAL and the structure of the graph is defined by certain Boolean or linear mappings between the two power sets. In the remainder of this section we define the basic graph operators presently included in GRAAL.

An undirected *pseudograph* is a triple $G = (V, A, \varphi)$ consisting of a vertex (or node) set V, an arc set A, and an *incidence operator*

$$\text{(i) } \varphi \in B(A, V), \quad \text{(ii) } \varphi\{a\} \in P_1(V) \cup P_2(V), \quad \forall a \in A . \tag{2.4}$$

Thus, for any arc $a, \varphi\{a\}$ 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 Harary [8] and others, we speak of a *multigraph* if in (2.4) the condition (ii) is replaced by (ii') $\varphi\{a\} \in P_2(V)$, $\forall a \in A$. The unqualified term *graph* is used if, in addition to (ii'), the restricted mapping $\varphi\colon P_1(A) \to P_2(V)$ is one-to-one.

For any undirected pseudograph $G = (V, A, \varphi)$ the *star operator* is the Boolean mapping

$$\sigma \in B(V, A), \quad \sigma\{v\} = \{a \in A \mid v \in \varphi\{a\}\}, \quad \forall v \in V , \tag{2.5}$$

while the standard *boundary operator* ∂ and *coboundary operator* δ are defined as the linear mappings

$$\partial \in L(A, V), \quad \partial\{a\} = (|\varphi\{a\}| - 1)\varphi\{a\}, \quad \forall a \in A \tag{2.6}$$

$$\delta \in L(V, A), \quad \delta\{v\} = \{a \in \sigma\{v\} \mid |\varphi\{a\}| = 2\}, \quad \forall v \in V . \tag{2.7}$$

Hence, for any node $v, \sigma\{v\}$ is the set of all arcs of G which are incident with v. The boundary operator ∂ maps each arc into the set of its two endpoints, provided they are distinct, and otherwise into the empty set. Finally, $\delta\{v\}$ consists of all arcs incident with v excluding any self loops.

Other operators are possible and may be included later. Each one of these operators can be used in place of φ to characterize graphs of a specific type. In fact, note that σ has the properties

$$\text{(i) } \sigma \in B(V, A), \quad \text{(ii) } P_1(A) \subset \bigcup_{v \in V} \sigma\{v\} \tag{2.8}$$

$$\text{(iii) } \sigma\{u\} \cap \sigma\{v\} \cap \sigma\{w\} = \emptyset \text{ for any distinct } u, v, w \in V .$$

Conversely, if for any sets V, A the mapping σ satisfies (2.8), then σ is the star operator of the pseudograph $G=(V,A,\varphi)$, where

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

Correspondingly, since on a multigraph G we have $\partial\{a\}=\varphi\{a\}$, $a \in A$, and $\delta\{v\}=\sigma\{v\}$, $v \in V$, it follows that the incidence structure of any multigraph can be defined in terms of the boundary operator or the coboundary operator.

In a graph G each arc is uniquely determined by the two element set of its endpoints, and hence the arcs are losing some of their own identity. Accordingly, it is often expedient to work exclusively with the nodes. For this we introduce for a graph $G=(V,A,\varphi)$ the *adjacency operator*

$$(2.9) \qquad \alpha \in B(V,V), \quad \alpha\{v\} = \{u \in V \mid \exists\, a \in \sigma\{v\}, \varphi\{a\}=\{u,v\}\}, \quad \forall v \in V .$$

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. In fact, if V is any set and the mapping α satisfies

$$(2.10) \qquad \text{(i) } \alpha \in B(V,V), \quad \text{(ii) } v \notin \alpha\{v\}, \quad \forall v \in V$$
$$\text{(iii) } u \in \alpha\{v\} \text{ if and only if } v \in \alpha\{u\}, \quad \forall u,v \in V ,$$

then $G=(V,A,\varphi)$ with

$$A = \{\{u,v\} \in P_2(V) \mid u \in \alpha\{v\}\}$$
$$\varphi \in B(A,V), \ \varphi\{a\} = \{u,v\} \text{ if } a = \{u,v\}, \quad \forall a \in A$$

is a well-defined graph with α as its adjacency operator. We call (V,α) the *node form* representation of G.

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

$$(2.11) \qquad \varphi_+,\varphi_- \in B(A,V), \quad \varphi_+\{a\}, \quad \varphi_-\{a\} \in P_1(V), \quad \forall a \in A .$$

In other words, $\varphi_+\{a\}$ and $\varphi_-\{a\}$ 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

$$(2.12) \qquad \varphi \in B(A,V), \quad \varphi\{a\} = \varphi_+\{a\} \cup \varphi_-\{a\}, \quad \forall a \in A .$$

As in the undirected case we speak of a *directed multigraph* if $\varphi\{a\} \in P_2(V)$,

for all $a \in A$, and of a *directed graph* (*digraph*) if, in addition, $\varphi\colon P_1(A) \to P_2(V)$ is one-to-one.

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

$$\begin{aligned} &\sigma_+ \in B(V,A), \quad \sigma_+\{v\} = \{a \in A \mid v = \varphi_+\{a\}\}, \quad \forall v \in V, \\ &\sigma_- \in B(V,A), \quad \sigma_-\{v\} = \{a \in A \mid v = \varphi_-\{a\}\}, \quad \forall v \in V, \end{aligned} \tag{2.13}$$

and we introduce also the combined star operator

$$\sigma \in B(V,A), \quad \sigma\{v\} = \sigma_+\{v\} \cup \sigma_-\{v\}, \quad \forall v \in V. \tag{2.14}$$

Thus, $\sigma_+\{v\}$ consists of all the arcs beginning at v and $\sigma_-\{v\}$ of those terminating at 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 (2.8) have to hold for both σ_+ and σ_- and (iii) can be replaced by (iii') $\sigma_+\{u\} \cap \sigma_+\{v\} = \emptyset$, $\sigma_-\{u\} \cap \sigma_-\{v\} = \emptyset$ for any $u \neq v$ in V.

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

$$\begin{aligned} &\partial_+, \partial_- \in L(A,V), \quad \partial_+\{a\} = \varphi_+\{a\}, \quad \partial_-\{a\} = \varphi_-\{a\}, \quad \forall a \in A, \\ &\delta_+, \delta_- \in L(V,A), \quad \delta_+\{v\} = \sigma_+\{v\}, \quad \delta_-\{v\} = \sigma_-\{v\}, \quad \forall v \in V. \end{aligned} \tag{2.15}$$

It is natural to define also the combined mappings

$$\begin{aligned} &\partial \in L(A,V), \quad \partial\{a\} = \partial_+\{a\} \triangle \partial_-\{a\}, \quad \forall a \in A, \\ &\delta \in L(V,A), \quad \delta\{v\} = \delta_+\{v\} \triangle \delta_-\{v\}, \quad \forall v \in V. \end{aligned} \tag{2.16}$$

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.

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

$$\begin{aligned} &\alpha_+, \alpha_- \in B(V,V) \\ &\alpha_+\{v\} = \{u \in V \mid \exists\, a \in \sigma_+\{v\},\ u = \varphi_-\{a\}\}, \quad \forall v \in V \\ &\alpha_-\{v\} = \{u \in V \mid \exists\, a \in \sigma_-\{v\},\ u = \varphi_+\{a\}\}, \quad \forall v \in V. \end{aligned} \tag{2.17}$$

Then the combined adjacency operator

$$\alpha \in B(V,V), \ \alpha\{v\} = \alpha_+\{v\} \cup \alpha_-\{v\}, \quad \forall v \in V \tag{2.18}$$

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 (2.10) 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 [10]. 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 (*a*-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 possible modules 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.

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 property variable requires an argument which is enclosed between the "dotted" left parenthesis '.(' and the right parenthesis ')' and should 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. 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 epxression*⟩ | ⟨*left part list*⟩ **empty**
⟨*left part list*⟩⟨*list expression*⟩ | ⟨*left part list*⟩ **nil**
⟨*left part list*⟩ ::= ⟨*left part list*⟩⟨*left part*⟩
⟨*left part*⟩ ::= ⟨*variable*⟩ := | ⟨*procedure identifier*⟩ := || ⟨*list identifier*⟩ :=
⟨*expression*⟩ ::= ⟨*arithmetic expression*⟩ | ⟨*designational expression*⟩
⟨*Boolean expression*⟩ || ⟨*set expression*⟩ | ⟨*list expression*⟩
⟨*function designator*⟩ ::= ⟨*procedure identifier*⟩⟨*actual parameter part*⟩ ||
⟨*list operator designator*⟩
⟨*set expression*⟩ ::= ⟨*set union*⟩ | ⟨*set expression*⟩ ~ ⟨*set union*⟩
⟨*set union*⟩ ::= ⟨*set sum*⟩ | ⟨*set union*⟩ ∪ ⟨*set sum*⟩
⟨*set sum*⟩ ::= ⟨*set intersection*⟩ | ⟨*set sum*⟩ △ ⟨*set intersection*⟩
⟨*set intersection*⟩ ::= ⟨*set primary*⟩ | ⟨*set intersection*⟩ ∩ ⟨*set primary*⟩
⟨*set primary*⟩ ::= ⟨*variable*⟩ | (⟨*set expression*⟩) | ⟨*function designator*⟩ |
⟨*subset designator*⟩ | ⟨*graph designator*⟩ | ⟨*a-set designator*⟩
⟨*subset designator*⟩ ::= **subset** (⟨*simple variable*⟩, ⟨*Boolean expression*⟩)
subset (⟨*simple variable*⟩ **in** ⟨*set expression*⟩, ⟨*Boolean expression*⟩)
⟨*a-set designator*⟩ ::= **create** | **create** (⟨*atom def list*⟩) |
atom (⟨*simple arithmetic expression*⟩)
⟨*atom def list*⟩ ::= ⟨*atom definition*⟩ | ⟨*atom def list*⟩, ⟨*atom definition*⟩
⟨*atom definition*⟩ ::= ⟨*property identifier*⟩ : ⟨*arithmetic expression*⟩ |
⟨*property identifier*⟩ : ⟨*Boolean expression*⟩ |
⟨*property identifier*⟩ : ⟨*set expression*⟩
⟨*graph designator*⟩ ::= ⟨*basic graph operator*⟩ (⟨*graph identifier*⟩) |
⟨*structure operator*⟩ (⟨*graph identifier*⟩, ⟨*set expression*⟩)
⟨*basic graph operator*⟩ ::= **arcs** | **nodes**
⟨*structure operator*⟩ ::= **adj** | **padj** | **nadj** | **inc** | **pinc** | **ninc** | **star** | **pstar** |
nstar | **bd** | **pbd** | **nbd** | **cob** | **pcob** | **ncob**
⟨*list expression*⟩ ::= ⟨*list element*⟩ | ⟨*list expression*⟩ ∘ ⟨*list element*⟩
⟨*list element*⟩ ::= ⟨*number*⟩ | ⟨*variable*⟩ | ⟨*list identifier*⟩ |
(⟨*expression*⟩) | ⟨*function designator*⟩
⟨*list operator designator*⟩ ::= ⟨*list operator*⟩ (⟨*list expression*⟩)
⟨*list operator*⟩ ::= **f** | **fd** | **l** | **ld**

EXAMPLES.

$S := X \cup Y \cap C \sim D \triangle M$; $L :=$ **bd**$(G,X) \cap$ **cob**(G,Y);
$M :=$ **nodes**$(G) \cup$ **arcs**(G);
$S :=$ **subset** $(x$ **in star**(G,Y), $cap\cdot(x) > 0)$; $S := S \cup$ **create**;
$x :=$ **create** $(name\colon i,\ cap\colon k)$; $X :=$ **atom** $(1) \cup$ **atom** $(2) \cup$ **atom** $(i+1)$;
$S :=$ **subset** $(x,\ cap.(x) > 0 \wedge cap.(x) < 20)$; $L := L \circ a \circ 3 \circ (a+b) \circ cap.(x)$

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 (or modules) of graphs can be distinguished by the data structure used to represent them and/or by the family of graph operators provided with this structure. The specific set of modules available depends upon the implementation. For example, four modules which are readily distinguishable in terms of their graph operators 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 operators **inc**, **pinc**, **ninc**; the star operators **star**, **pstar**, **nstar**; the boundary operators **bd**, **pbd**, **nbd**; the coboundary operators **cob**, **pcob**, **ncob**; the adjacency operators **adj**, **padj**, **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 ($\sim$), union ($\cup$), symmetric sum ($\triangle$), 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 the 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 statement*⟩ ::= ⟨*assignment statement*⟩ | ⟨*go to statement*⟩ | ⟨*dummy statement*⟩ | ⟨*procedure statement*⟩ || ⟨*assign statement*⟩ | ⟨*detach statement*⟩

⟨*assign statement*⟩ ::= **assign** (⟨*graph identifier*⟩, ⟨*node*⟩) | **assign** (⟨*graph identifier*⟩, ⟨*node*⟩−⟨*node*⟩) | **assign** ((⟨*graph identifier*⟩, ⟨*node*⟩−⟨*node*⟩ **to** ⟨*arc*⟩)

⟨*detach statement*⟩ ::= **detach** (⟨*graph identifier*⟩) | **detach** (⟨*graph identifier*⟩, ⟨*set expression*⟩) | **detach** (⟨*graph identifier*⟩, ⟨*set expression*⟩−⟨*set expression*⟩)

⟨*node*⟩ ::= ⟨*variable*⟩ | ⟨*a-set designator*⟩

⟨*arc*⟩ ::= ⟨*variable*⟩ | ⟨*a-set designator*⟩

EXAMPLES.

assign (G, $n1-n2$ **to** $a1$); **assign** (G, $n-m$); **assign** (G, n); **detach** (G); **detach** (H, $S \cup T$); **detach** (H, $S-T$)

Semantics.

The assign statement constructs the incidence structure of a graph. It appears in three forms: one assigns a node to a graph; another assigns a pair of nodes to a graph in node-form, with the connecting arc implied; and a third assigns a pair of nodes as end points to the specified arc. If the graph is directed, the first node specified is the initial node. An error results if a previous assign statement is contradicted, an atomic element is assigned as a node and an arc in the same graph, or a non-atomic set is assigned as a node or arc.

The detach statement removes elements from a graph. It appears in three forms. One detaches all nodes and arcs of a graph. Another detaches

all the elements of a specified set from a graph (if the element is an arc, it is simply removed; if it is a node, it is removed along with all arcs incident with it). A third form detaches all arcs connecting two specified sets of nodes. If the graph is directed, the first set is taken as the set of initial nodes.

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 statement*⟩
⟨*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*⟩) | ⟨*label*⟩ : ⟨*removal statement*⟩

EXAMPLES.

for all x **in** $X \cap Y$ **do if** *capacity*.$(x) > 0$ **then** $M := M \cup x$;
for all i **in** *List* **do** $n := n+1$; **while** $\neg(S \subseteq T)$ **do** $T := S$;
delete (**nodes** $(G) \cup$ **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

$X \subseteq Y$; $X = Y$; $Y \neq$ **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.

II. *Standard Functions*

Add to the list of standard functions:

size (T)	Number of elements in the set or list T
parity (T)	True if size (T) is odd, else false, where T is a set or list
index (x, T)	Index of the place taken by the element x in the set or list T
elt (i, T)	The ith element of the set or list T
check $(f(S))$	True if the property variable f is defined on each element of the set S and false otherwise
count (S)	Sequence number of a given a-set S. If S is not an a-set, the sequence number of its first element is returned.
maxcount	Sequence number of the last a-set created

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

4. Examples of GRAAL programs.

We present now several typical graph algorithms in the form of GRAAL procedures. The principal aim here is to illustrate some of the main features of the language without attempting at this time 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* $1 \leqq k \leqq m$, $1 \leqq i, j \leqq n$ *and signifies that the k-th arc has the i-th node as initial, and the j-th 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
procedure readtwo (G, name);
graph G; alpha property name;
comment A procedure 'read (buffer)' is assumed to be available which al-
  lows 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 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 size(buffer) = 0 then assign (G, x)
                        else while size(buffer) ≠ 0 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 pseudograph G.

```
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 (G, x), inc (G, a) ⊆ N);
    N := N ~ x;
    if S = empty then assign (SubG, x) else for all a in S do
    begin y := inc(G, a) ~ 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 (G, x);
    for all a in S do
    begin if x = inc (G, a) 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 representing 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 size(L) ≠ 0 do
  begin S := fd(L); x := create (ref: S); assign (ConG, x);
    for all T in L do
    if inc (G, star(G, S)) ∩ 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.

```
procedure cocycles (G, C);
graph G; set list C;
comment This procedure determines a basis for the cocycle 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 (G, S) ~ A; S := inc(G, A) ~ 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 (G, u);
  while T ≠ empty do
  begin for all a in T do
    begin w := bd(G, a); y := w ~ S;
      if y ≠ empty then
      begin S := S ∪ y; x := w ~ y; assign (Tree, x−y to a) end
    end;
    T := cob (G, S)
  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 com-
  ponents of G. From this spanning tree the procedure generates, in a
  standard manner, a basis for the cycle space of the particular component;
begin set X, S, T, a;
  X := star (G, nodes(Tree)) ~ arcs (Tree);
  for all a in X do
  begin S := a; T := inc(G, a);
    if size(T) ≠ 1 then
    while T ≠ empty do
    begin T := ncob (Tree, T);
      if T ≠ empty then
      begin S := S △ T; T := pbd (Tree, T) 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 com-
  ponent of G, and from 'Tree' the 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 (Tree, T); T := pcob (Tree, S) ~ T end;
  l: Cuts := Cuts∘cob (G, S);
  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** (G, S) 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 [13] 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 a, b, smin, tmin; boolean flag;
  real property sdist, tdist; set property in, out;
  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 there-
    from 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;
  m := smin := tmin := sdist.(start) := tdist.(term) := 0;
  S := SR := start; T := TR := term; flag := false;
comment Insert a fictitious arc w from start to term;
  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 := nil;
for all x in SR do
```

```
    if sdist.(x) < m then begin m := sdist.(x); path := x end
                     else if sdist.(x) = m then path := x∘path;
    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(G, x) do
      begin y := nbd(G, z); 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 := nil;
    for all x in TR do
    if tdist.(x) < m then begin m := tdist.(x); path := x end
                     else if tdist.(x) = m then path := x∘path;
    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(G, x) do
      begin y := pbd(G, z); 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;
  comment No path;
  nopath: path := nil; go to exit;
  comment Breakthrough, establish 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 := x := y; path := u;
    while x ≠ start do
```

```
    begin z := in.(x); x := pbd(G, z); path := x∘z∘path
    end;
  while y ≠ term do
    begin z := out.(y); y := nbd(G, z); path := path∘z∘y
    end;
comment Remove fictitious arc and exit;
  exit: remove(w)
end
```

REFERENCES

1. S. Chase, *Analysis of algorithms for finding all spanning trees of a graph*, Department of Computer Science Report 401, Univ. of Illinois, Urbana, Illinois, 1970.
2. D. Childs, *Feasibility of a set-theoretic data structure—a general structure based on a reconstituted definition of a relation*, Proc. IFIP Congress 68 (1968), 162–172.
3. D. Childs, *Description of a set-theoretic data structure*, Proc. Fall Joint Computer Conference 68 (1968), 557–564.
4. S. Crespi-Reghizzi and R. Morpurgo, *A graph theory oriented extension of ALGOL*, Calcolo 5 (1968), 1–43.
5. S. Crespi-Reghizzi and R. Morpurgo, *A language for treating graphs*, Comm. ACM 13 (1970), 319–323.
6. D. Friedman, *GRASPE graph processing: a LISP extension*, Computation Center Report TNN-84, Univ. of Texas, Austin, Texas, 1968.
7. D. Friedman, D. Dickson, J. Fraser, and T. Pratt, *GRASPE 1.5, a graph processor and its application*, Department of Computer Science Report RS1–69, Univ. of Houston, Houston, Texas, 1969.
8. F. Harary, *Graph Theory*, Addison–Wesley, Reading, Massachusetts, 1969.
9. R. Hart, *HINT: a graph processing language*, Institute for Social Science Research Technical Report, Michigan State Univ., East Lansing, Michigan, 1969.
10. P. Naur (ed.), *Revised report on the algorithmic language ALGOL 60*, Comm. ACM 6 (1963), 1–17.
11. J. Nievergelt, *Software for graph processing*, SIGSAM Bulletin No. 14, 1970.
12. J. Pfaltz, *TREETAN—A FORTRAN IV subroutine package for manipulation of rooted trees*, Computer Science Center Technical Report 65-23, Univ. of Maryland, College Park, Maryland, 1965 (revised 1970).
13. I. Pohl, *Bi-directional and heuristic search in path problems*, Computer Science Department Technical Report CS-136, Stanford Univ., Stanford, California, 1969.
14. T. Pratt, and D. Friedman, *A language extension for graph processing and its formal semantics*, Comm. ACM 14 (1971), 460–467.
15. R. Read, C. King, C. Cadogan, and P. Morris, *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, 1969.
16. R. Read, *Teaching graph theory to a computer*, in "Recent Progress in Combinatorics", Academic Press, New York, New York, 1969.
17. R. Tabory, *Premiers elements d'un langage de programmation pour le traitement en ordinateur des graphes*, in "Symbolic Languages for Data Processing", Gordon and Breach, New York, New York, 1962.

18. M. Wolfberg, *An interactive graph theory system*, Moore School of Electrical Engineering Report 69-25, Univ. of Pennsylvania, Philadelphia, Pennsylvania, 1969.
19. M. Wolfberg, *An interactive graph theory system*, Technical Report CA-7003-0211, Massachusetts Computer Associates, Wakefield, Massachusetts, 1970.

COMPUTER SCIENCE CENTER
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
USA