SETL Newsletter # 115

A SETL Representation

of the Maryland

GRAAL graph-munipulation language

G. Weinberger

A. Tenenbaum

August 21, 1973

A graph algorithmic language (GRAAL) has been developed by Rheinhold, Basili, and Mesztenyi at the University of Maryland to describe and implement graph algorithms. The language provides for several non-standard data structures with attendant operations. Prominent among these is the GRAPH with about twenty primitive operators for graph manipulation. Also included is the data type SET with the standard set operations, and a data type for stacks.

The documentation analyzed consisted of

1. A description of a theoretic basis for a graph algorithmic language.
2. A description of an implemented version (as an extension of Fortran) called FGRAAL.
3. A text describing some 40 graph algorithms written in FGRAAL.
4. An article defining a proposed graph algorithmic extension of ALGOL 60.

This analysis was done with a view towards extracting the fundamental problems of computer representation of graphs and the dictions appropriate to their manipulation. An analysis was also made of the use of SETL for coding graph algorithms.

This newsletter proposes to do the following:

1. Describe some key issues presented by the GRAAL implementations.

2. Discuss the issue of graph representation in SETL.
3. Describe a SETL implementation of a complete set of primitive operations for graph manipulation.
4. Describe a SETLB encoding of the algorithms presented in FGRAAL Algorithms and present possible ways of using SETL's power to improve the algorithms.
5. Present some conclusions comparing SETL and GRAAL.

I. Some notes on GRAAL:

One is directed to the GRAAL documentation [see the bibliography] for a complete description of the language. In this section, we shall present some of the design decisions made by the GRAAL group, and the effect of these decisions upon the language.

GRAAL exists in several versions. It is not entirely clear whether a sharp demarcation exists between the reference and implemented versions of the language, and whether, as the case in SETL, the implemented version is only meant as a first approximation to the language itself. It would seem, however, that the language is thought of as a set of several implementations each

designed as an extension of a pre-existing higher level language. For example, one implemented version called FGRAAL is an extension of FORTRAN and follows the FORTRAN style very closely. Another proposed version exists which is an extension of Algol.

The concept of GRAAL as an extension of a pre-existing language, has forced (or perhaps suggested) several interesting, although theoretically unappealing features. For example, in FGRAAL a set is thought of as an array, each element being given a sequence number. Thus one can specify the "first" member of a set. However, the union of two sets is not the concatenation of the two arrays, but rather the true set-theoretic union. Further, the elements of a set are not viewed as individual elements but rather as "atomic" or single element sets. Thus suppose one has a set S= {1,3,5} and one executes

```
A=ELT(1,S)          (first element of S)
```

A is set equal to the atomic {1} . One can then execute

```
S=S.DIFF.A          (where .DIFF. is the FGRAAL
                     operator for set difference)
```

to set S= {3,5} . Thus, the FGRAAL .DIFF. operator can be used both as the SETL set difference operator and the less operator which removes an element from a set. Similarly the .UN. operator acts both as the set union (+ applied to sets) and as the with operator.

Although this facility is confusing at first, it is

useful in writing short code in which on different loop iterations the second operand is alternately a single element and a set. It also adheres to the principle of reusing syntactic space whenever possible, although in a manner different from SETL. It is, of course, a quite simple matter to achieve the same effect in SETL. For example, we could define an operator UN. for union as follows:

```
definef A UN.B
return if atom A then
                    if atom B then {A,B}
                              else B with A
                else if atom B then A with B
                              else A+B ;

end;
```

The internal representation of graphs in FGRAAL, although meant to be transparent to the user, forces certain annoying restrictions. Arc and node constants must be of type INTEGER and each graph comes supplied with a 'sequence number' which is needed (it seems) to distinguish it from other graphs. The user can delete or add nodes, arcs, or arc-node pairs to a graph by invoking various primitive functions. However, since one cannot manipulate the internal structures, one is limited in the types of operations that one can perform.

II. Graph representation in SETL:

In developing graph algorithms in SETL, the first issue to be faced is a SETL representation for a graph. This question is closely akin to the problem which a FORTRAN programmer would face in deciding what representation to use for a graph. Several possibilities suggest themselves:

1. Graphs by pairs: Since an arc consists of two adjacent nodes, a graph can be represented as a set of pairs. Thus for example, the graph

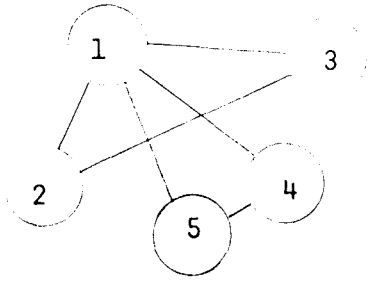

Fig. 1

could be represented by

(1) {{ 1,2} , {1,4} ,{ 1,5} ,{ 1,3} ,{ 2,3} ,{ 4,5}}

However this does not allow for a convenient representation of directed graphs in which an arc consists not only of a pair of nodes but also of an orientation between them. To solve this problem, the graph of Fig. 2

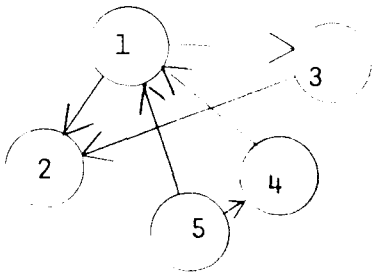

Fig. 2

would be represented by a set of ordered pairs, e.g.:

(2) { <1,2> , <1,3> ,< 3,2> , <4,1> ,< 5,1> ,< 5,4>}

If representation (2) is chosen, we would also like to represent the unordered graph of Fig. 1 as a set of ordered pairs for the sake of uniformity. Two possibilities present themselves. The first is to consider an unoriented arc between two nodes, as consisting of two oriented arcs. Thus the graph of Fig. 1 would be represented by:

(3) {< 1,2 > , < 2,1 > , < 1,4 > , < 4,1 > , < 1,5 > , < 5,1 > ,
< 1,3 > , < 3,1 > , < 2,3 > , < 3,2 > , < 4,5 > , < 5,4 >}

The problem with such a representation is that it takes the approach of forcing the explicit specification of the two possible orientations and thus wastes space. Further in processing an undirected graph, it is often desirable to process each unoriented arc only once, thus necessitating a test each time an arc is a candidate for processing, whether or not its mirror image has already been processed.

The alternative is to represent both the graphs of Fig. 1 and Fig. 2 by (2) and force the programmer to keep track of whether he is dealing with an undirected or directed graph. This does not produce as great a burden as would be imagined since in a specific algorithm, one is dealing with a specific type of graph (directed or undirected). Of course, in the case of a graph with both directed and undirected arcs, the graph would be represented as directed with an undirected arc being duplicated by its reversal. If a programmer wanted to apply a subroutine which expects a directed graph to an undirected graph, he could use a conversion routine to convert representation (2) to (3):

```
definef convertpairs (undgr);
        dgr=nl;
        (∀t ε undgr) dgr=(dgr with t) with  < t(2),t(1) > ;
        return dgr;
end convertpairs;
```

Incidentally, FGRAAL handles the orientation question quite nicely by associating a "property" with a graph. Thus, if G is declared to be graph (following FORTRAN, a name must be declared as naming a specific type of item), then associated with G is the true-false property ORIENT(G) which is 1 if the graph is directed and 0 otherwise. Since the name G refers to a specific graph, even if the value of G changes (i.e. an arc is added or deleted), the ORIENT property remains unchanged. This feature is similar to the PL/I structure. To implement such a property in SETL, one would have to represent a graph g as an ordered pair, the first element being the graph itself, the second being either true or false depending on whether you wished to consider the graph as directed or undirected. This would probably add an unnecessarily high degree of clumsiness to the graph algorithms.

Unfortunately, the scheme of representation by pairs while sufficient for most graphs is not sufficiently flexible for a total graph algorithms package. This is because a graph need not be restricted to allowing only a single arc between two nodes. Thus the undirected graph of Fig. 3

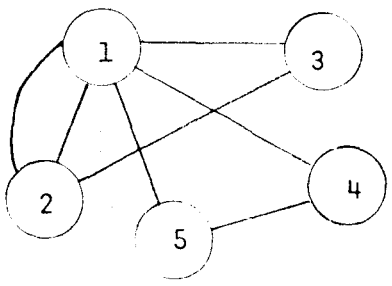

Fig. 3

would be multigraph in which two arcs connect nodes 1 and 2 and as such would be distinct from the graph of Fig. 1 in which only one arc connects these two nodes. Clearly, representation by pairs is insufficient for such a graph since in that representation an arc is defined by its two end nodes and no provision can be made for the same two nodes to define two or more different arcs. This problem arises becauses the nodes are given names (e.g. 1,2...) in the repesentation, but arcs are not. Instead, an arc is viewed not as a distinct entity but as a composite of two nodes. An alternative representation which would solve the problem is representation by triples.

2. Graphs by triples: In this representation, which was chosen for the SETL implementation, the arcs as well as the nodes are given names. Thus the graph of Fig. 1 would be viewed as

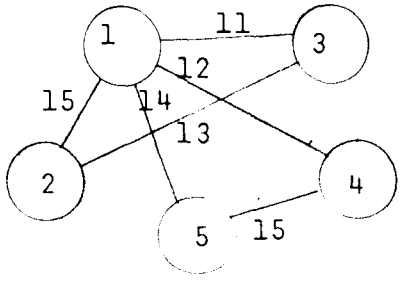

Fig. 4

and would be represented as a set of triples, e.g.:

{< 16,1,2> ,< 11,1,3> ,< 12,1,4> ,

< 14,1,5> ,< 13,2,3> ,< 15,4,5>}

The first element of each triple is the name of the arc represented by that triple and second and third elements, the names of the two nodes which that arc connects. In the case of a directed arc, the arc is thought of as

going from the node in the second element to the node in the third. The graph of Fig. 2 would be viewed as

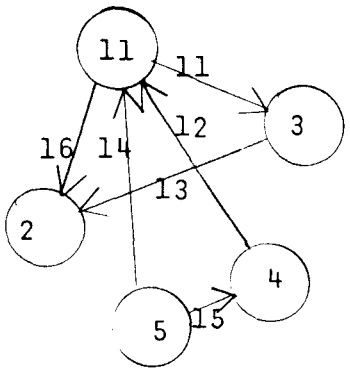

Fig. 5

and would be represented by

{< 16,1,2 > , < 15,5,4 > , < 11,1,3 > ,
< 12,4,1 > , < 14,5,1 > , < 13,3,2 >} .

Multigraphs are easily accomodated in this system of representation. The graph of Fig. 3 would be viewed as in Fig. 6

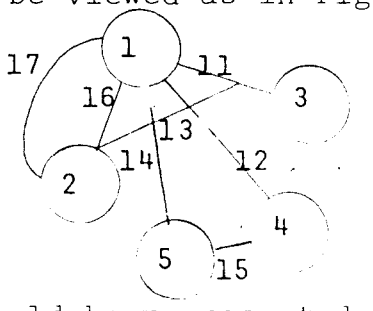

Fig. 6

and would be represented by,

{< 17,1,2 > , < 16,1,2 > , < 14,1,5 > ,
< 12,1,4 > , < 11,1,3 > , < 13,2,3 >} ,
< 15,4,5 >} .

Note that although both arcs 16 and 17 connect nodes 1 and 2, they are represented as the two distinct arcs that they really are in the multigraph.

One problem, however, remains to be solved and that is the problem of isolated nodes. Both representation by pairs and triples present a graph as a set of arcs. However, a graph may have a node which is not adjacent to any arc and which would therefore not be included in the representation. For example, consider the graph of Fig. 7

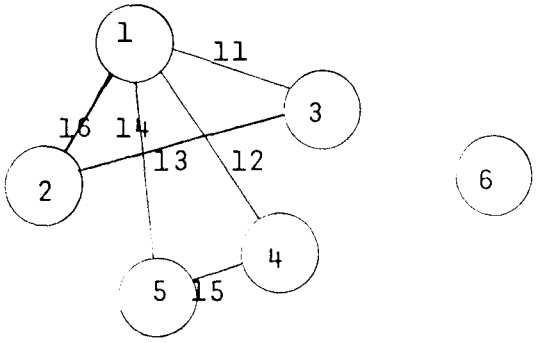

Fig. 7

which contains 6 arcs none of which include node 6. To remedy this we adopt the following convention; an isolated node X will be represented by the triple

<Ω,X,Ω> where Ω is the undefined atom in SETL (OM. in SETLB).

Thus, the graph of Fig. 7 is represented by

{<16,1,2> , <11,1,3> , <14,1,5> ,
<12,1,4> , <13,2,3> , <15,4,5> ,
<Ω,6,Ω>} .

Although this involves occasional checking for Ω , it does allow representation of graphs and directed graphs in full generality and is fairly easy to work with.

We have also provided for two possible triple representations of node graphs (i.e. graphs defined

solely in terms of their nodes). In the standard version, each node pair <X,Y> is represented by the triple <Ω,X,Y> . The graph of Fig. 1 above would then be defined as follows:

{<Ω,1,2> , <Ω,1,4> , <Ω,1,5> ,
<Ω,1,3> , <Ω,2,3> , <Ω,4,5>} .

In the numbered version an arbitrary sequential numbering is assigned to the arcs. The same graph would then be represented by

{<1,1,2> , <2,1,4> , <3,1,5> ,
<4,1,3> , <5,2,3> , <6,4,5>} .

This will be more fully defined in the conversion routines described below.

Other methods of graph representation are possible. GRAAL provides Input-Output routines for reading and writing graphs expressed in one of the various forms described below. The SETL implementation expands this by allowing one to manipulate the structure which represents a graph, thus enabling the encoding of an algorithm using the method of graph representation which is most convenient and using conversion routines to put the graph into the desired form.

3. Graphs by stars: In this representation, a graph is represented by a set of ordered pairs, the first element of each pair being a node and the second, the

set of arcs incident on the node. For example, the graph of Fig. 7 would be represented by

{<1, {11,12,14,16}> , <2, {13,16}> ,
<3, {11,13 } > , <4, {12,15} > ,
<5, {14,15 } > , <6,nl > } .

This method takes care of the isolated node problem with minimal disruption. Perhaps the greatest objection to using this as a standard representation is that it is counter-intuitive. A graph is usually not thought of as a collection of nodes from each of which a set of arcs emanates, but rather as a set of arcs interconnecting a set of nodes. This counter-intuition problem would probably make it more difficult to program using such a representation. A SETL objection is that to get at an arc one has to go into a high level of nesting (e.g. an element of a set which is itself an element of a tuple which is an element of a set). This, of course, is another obstacle to neat and easy programming which does not exist in the pair and triple representations.

Again, some modification must be made in the representation by stars to allow for directed graphs. Various possibilities suggest themselves. FGRAAL, which provides only an I/O representation by stars, represents the graph by a node followed by arcs printed as positive or negative integers. For example, the pair < 1, { 16, -14,-12,11 } > would be an element in the star repre-

sentation of Fig. 5. This suffers from the general objection against forcing nodes and arcs to be represented as positive integers. We have chosen instead to represent directed graphs by stars as a set of triples the first element being a node, the second element, the set of arcs leaving the node, the third, the set of arcs entering the node. The graph of Fig. 5 would be represented by

{<1, {16,11} , {14,12} > , <2, nl , {16,13}> ,
<3, {13} , {11} > , <4, {12} , {15} > ,
<5, {14,15} > } .

Still another method with much the same advantages and disadvantages of representation by stars is representation by adjacency.

4. Graphs by adjacency: In this method, which is of primary importance in node-graphs, a graph is also represented as a set of ordered pairs. The first element of the pair is a node and the second element, the set of nodes which are connected to the given node by some arc. Thus the graph of Fig. 7 would be represented by

{<1, {2,3,4,5} > , <2, {1,3 } > , <3, {1,2} > ,
<4, {1,5 }> , <5, {1,4} > , <6,nl > } .

Directed graphs can be treated by a technique similar to the one classified for STARS above. The graph of

Fig. 5 would appear as

{<1, {2,3} , {5,4}> , <2,nl, {1,3}> ,
<3, {2} , {1}> , <4, {1} , {5}> ,
<5, {1,4} ,nl > } .

where the first element of each triple is a node, the second, the set of nodes leaving the given node (PADJ), the third, the set of nodes entering the node (NADJ).

5. Conversion routines: The above sections described various forms of graph representation. As noted earlier, representation by triples was chosen as the basic form for the SETL implementation of the graph algorithms. However, a set of routines are provided for converting from any of the other forms (pairs, stars, and adjacency) to triples and vice-versa. These routines, which were quite simple to create, display the facility of manipulating graphs in SETL.

The GRAAL conversion routines are provided only for I/O, that is, they allow one to either print or read the graphs in various formats. The SETL routines are written as functions and can be used to translate from one form to another. The following tables identify the routines.

Table 1. Functions to translate from alternate form to triples.

Routine Name	Input Parameter	Returns
RDGPRS	pairs	triples with Ω arcs
RDGPRSN	pairs	triples with arbitrarily numbered arcs
RDPADJUD	undirected adjacency	pairs
RDPADJD	directed adjacency	pairs
RDGADJUD	undirected adjacency	triples with Ω arcs
RDGADJUDN	undirected adjacency	triples with arbitrarily numbered arcs
RDGADJD	directed adjacency	triples with Ω arcs
RDGADJDN	directed adjacency	triples with arbitrarily numbered arcs
RDGSTUD	undirected stars	triples
RDGSTD	directed stars	triples

These routines which are inherently set-theoretic, are easily expressed. As an example, consider the translation from undirected adjacency to triples (RDGADJUD). We first use the routine RDPADJUD to transform the graph to pair notation. The input is a set, S, in the form expressed above (Section 2.4). An element $p \in S$ is of the form $\langle n, \{V_1,\ldots,V_j\}\rangle$ where n is a node and $V_1,\ldots,V_j$ are the nodes adjacent to n. The graph, G, created must contain all pairs $\langle n,V_i\rangle$. However, since the graph is undirected, we do not want both $\langle n,V_i\rangle$ and $\langle V_i,n\rangle$ in G. The routine is therefore of the following form:

```
definef rdpadjud(s);
local g,p,v;
g=nl;
(∀p ∈ s)
if p(2) eq nl then ( < Ω,p(1),Ω > ) in g;
/*if the node is not adjacent to any other nodes add
< Ω,n,Ω>   to g*/
                else g=g + {<p(1),v> , v ∈ p(2) | <v,p(1)> ∉ g}
/*add all pairs <n,v> if <v,n> is not already there*/
return g;
end rdpadjud;
```

This routine transforms the graph to pairs. To complete the transformation into triples, we use the routine RDGPRS.

```
definef rdgprs(S)
return     {<Ω ,p(1),p(2)>     ,p∈ S >}   ;
end rdgprs;
```

The final routine to transform adjacency to triples calls on both of the above routines:

```
definef rdgadjud(S);
return rdgprs (rdpadjud(S));
end rdgadjud;
```

The routines to convert from triples to the various other forms are listed below.

Table 2.

Routine Name	From triples to
WRGPRS	pairs
WRGSTD	stars, for a directed graph
WRGSTUD	stars, for an undirected graph
WRGADJD	adjacency for a directed graph
WRGADJUD	adjacency for an undirected graph

These routines are easily expressed since they use the primitive functions for graph manipulation (described below in Section 3). e.g.

```
definef wrgadjd(G);
/* to change from triples to adjacency representation
for a directed graph */
return     {< n,padj(G,n), nadj(G,n) >  ,
           n ∈  nodes(G) }   ;
end wrgadjd;
```

III. Basic graph operations in SETL:

FGRAAL provides for a large set of operations on graphs which are used as FORTRAN subroutine calls. The implementation of these operations are invisible to the user. For example, given a graph G and a subset A of arcs of the graph, one would want a routine which returned the set of nodes incident (connected) to one of the arcs in A. FGRAAL provides this facility with the incidence operator and one would code

N=INC(G,A).

Using the extended triple representation discussed in (2) above, the SETL routine for INC would be:

```
definef inc(g,a);
local z,i;
return if atom a then  {g(a)(1),g(a)(2)}
                else   {g(z)(i),z ∈a, 1≤ i≤ 2}  ;
end inc;
```

Given the graph g of Fig. 7:

{<16,1,2> , < 14,1,5> , < 12,1,4> , < 11,1,3> ,

<13,2,3> , < 15,4,5> , < Ω,6,Ω > }.

and the set of arcs a= { 14,12,15 } ,

inc(g,a)= {1,5,4} .

Note that in GRAAL, a single element of a set is indistinguishable from a singleton set so that one can call INC even if A is a single arc. Thus, INC acts to some extent as a generic routine. To imitate this facility in the SETL routine, a test is made as

to whether or not the second argument is a set and if not the routine simply returns the endpoints of the arc. Thus inc(g,12)= {1,4 } , where g is the above graph.

Two very common operations on a graph are finding the nodes and arcs of a given graph. In SETL these are coded as follows:

```
definef arcs(g); local z;
return   {z(1),z ∈ g |z(1) ne  Ω } ;
end arcs;

definef nodes(g); local z,i;
return   {z(i),z ∈ g, 2 <i <3 |z(i) ne  Ω}   ;
end nodes;
```

For g as in Fig. 7, nodes(g)= { 1,2,3,4,5,6 } and arcs(g)= {11,12,13,14,15,16} .

The routine star is given a graph and a set of nodes (or a single node) and returns the set of all arcs which are incident to some node in the set.

```
definef star(g,n); local z,j;
j=if atom n then { n}  else n;
return  { z(1),z ∈ g |z(1) ne Ω and ({ z(2),z(3)} *j) ne nl};
end star;
```

The routine adj is given a graph and a set of nodes. It returns the set of all nodes which are connected to some node in the input set by a single arc of the graph.

```
definef adj(g,n); local p,a;
return if atom n then {p→nodes(g)| (∃ a ∈ star(g,n)|
                                    inc(g,a) eq {p,n})}  ;
                  else [+:p∈ n] adj(g,p);
end adj;
```

To illustrate:

star(g, { 5,2 })= { 13,16,14,15 }

adj(g, { 5,2 })= { 1,3,4 }

where g is the graph of Fig. 7.

Note the use of the compound operator in defining adj. The compound operator facility of SETL gives these algorithms a clear and concise expression which would otherwise be lacking.

Directed graphs can be operated on by the preceding operators, but these operators take no account of the orientation of the arcs. It is often desirable to specify such sets as "the set of arcs leading outward from a given node" or "the set of nodes in which a given set of arcs terminates". To facilitate such concepts, GRAAL provides a positive and negative form of the above operators. Thus, given a graph g and a set of arcs a, pinc(g,a) gives the set of nodes from which some arc of a emanates, while ninc(g,a) gives the set of nodes in which some arc of a terminates. Similarly, pstar(g,n) gives the set of arcs which have a node in n as a starting point and nstar(g,n) gives the set of arcs which have a node in n as a terminating point. Corresponding to the adjacency operator we have padj(g,n) which gives the set of nodes n' such that there is an arc in g going from a node in n to a node in n' and nadj(g,n) which gives the set of nodes n' such that there is an arc in g going from a node in n' to a node in n.

To illustrate with the graph g of Fig. 5:

pinc(g, {14,12,15}) = {5,4} , ninc(g, {14,12,15})= {1,4}

pstar(g, {5,2}) = {15,14} , nstar(g, {5,2}) = {13,16}

padj(g, {5,2}) = {1,4} , nadj(g, {5,2}) = {1,3}

The following is SETL code for the routines PINC, NINC, PSTAR, NSTAR, PADJ, and NADJ:

```
definef pinc(g,a); local p;
return if atom a then {g(a)(1)}
                 else {g(p)(1), p∈a} ;
end pinc;


definef ninc(g,a); local p;
return if atom a then {g(a)(2)}
                 else {g(p)(2), p∈a} ;
end ninc;


definef pstar(g,n); local z,j;
j= if atom n the {n} else n;
return {z(1),z∈g | z(1) ne Ω and z(2)∈j} ;
end pstar;


definef nstar(g,n); local z,j;
j= if atom n then {n} else n;
return {z(1),z∈g | z(1) ne Ω and z(3)∈ j} ;
end nstar;


definef padj(g,n); local p,a;
return if atom n then{p∈nodes(g)| (∃ a∈ pstar(g,n)|
                                    ninc(g,a) eq {p} )}
                 else  [+:p∈n] padj(g,p);
end padj;


definef nadj(g,n); local p,a;
return if atom n then{p∈nodes(g)|  (∃ a∈ nstar(g,n)|
                                     pinc(g,a) eq {p} )}
                 else [+:p∈n] nadj(g,p);
end nadj;
```

Note that each of these three operator categories: inc, star, and adj are expressed as unions over a set and that

inc(g,a)= pinc(g,a) + ninc(g,a)

star(g,n)=pstar(g,n) + nstar(g,n)

adj(g,n)= padj(g,n) + nadj(g,n)

There is another class of operators whose members are expressed not as a union over a set but as the symmetric difference over a set. The first of these is the coboundary of a set of nodes in a graph. This is the set of all arcs which connect the set to a node outside the set. Thus, if g is the graph of Fig. 7 and n={6,1,6} then cob(g,n)={14,11,16,15} . Mathematically, cob is defined recursively on the number of elements in the node set. If n is a single node then cob(g,n)={a $\in$ star(g,n) | #inc(g,a)=2} . This last condition on a is to exclude a self-loop from cob. Finally, if n is a set of nodes, then cob(g,n) is the symmetric difference over all nodes n_0 in n of cob(g,n_0).

Similarly, given a set a of arcs in a graph, the boundary of a is defined as the set of nodes which are incident at only one point (i.e. again excluding a node for which an arc is a self-loop) to exactly one of the arcs in the set. Thus if g is, again, the graph of Fig. 7 and a={11,12,13,14} , then bd(g,a)= {5,4,2} . Mathematically, if a is a single arc,

$$bd(g,a)= \begin{cases} inc(g,a) \text{ if } \#inc(g,a)=2 \\ \underline{nl} \text{ otherwise} \end{cases}$$

and if a is a set of arcs bd(g,a) is the symmetric difference of bd(g,a_0) for all $a_0 \in$ a.

The publication versions of the operators cob and bd follow: [1]

```
definef cob(g,n);
local p,d;
return if atom n then star(g,n)-{ d ∈ arcs(g) | <d,n,n> ∈ g}
                 else [/:p∈ n] cob(g,p);
end cob ;
```

```
definef bd(g,a);
local p;
return if atom a then if #inc(g,a) eq 2 then inc(g,a)
                                        else nl
                  else [/:p∈ a] bd(g,p);
end bd;
```

---

1 Note that whereas in publication SETL the symmetric difference operator is indicated by a /, in the current SETLB version it is indicated by //. However, due to a deficiency in the current SETLB front end, the operator // when used in a compound operator yields a syntax error (since the parser upon detecting a '[' expects an operator followed by a colon. It gets the operator but then gets the second / rather than the expected colon).

To remedy the situation, we defined a new user-defined operator P. by

```
definef A P. B
return  A//B
end;
```

and then rewrote the SETLB routine using the operator P. as the compound operator rather than the troublesome //. This led to the discovery of a SETLB error in translating a user defined operator in a compound form. However, this problem has been remedied and the currently running SETLB versions of the two algorithms are:

```
definef cob(g,n);
local p,d;
return if atom n then star(g,n)-{ d ∈arcs(g) | <d,n,n>∈g}
                 else [P.: p∈ n] cob(g,n);
end cob;
```

(continued on next page)

For directed graphs, we again introduce positive and negative versions of cob and bd. However, these versions do not bear the same relation to their antecedents as do pinc, ninc, pstar, nstar, etc. to inc, star, etc. For example, pcob is not defined as the set of arcs leading from the given set of nodes to some external node; rather it is defined as the set of arcs emanating from any of the nodes in the set. Similarly, ncob is defined as the set of nodes terminating in some one of the nodes in the set. This is because ncob and pcob are defined so that the relation cob=pcob//ncob holds. Thus cob is unlike inc, adj, and star where the relations

inc=pinc+ninc

adj=padj+nadj

star=pstar+nstar

hold.

Indeed, upon examination it is apparent that pcob is identical to pstar and ncob to nstar. However, cob represents an entirely different relation from star since

star=pstar+nstar=pcob+ncob

but cob=pstar//nstar=pcob//ncob.

---

1 (contd)

```
definef bd(g,a);
local p;
return if atom n then if#inc(g,a) eq 2 then inc(g,a)
                                       else nl
                  else [P.: p∈a] bd(g,p);
end bd;
```

Why GRAAL provides two distinct operator names for the same operator is not entirely clear, although it seems to stem from the graph theoretic origins of the language. Pcob is defined recursively, so that pcob(g,n) where n is an atomic set (a set with only one element) is identical to pstar(g,n) (i.e. the set of all arcs emanating from the node n) and pcob(g,n) where n is a general set, is defined as the symmetric difference over all atomic sets n_o in n of pcob(g,n_o) (which is the same as pstar(g,n_o)). However, since an arc can emanate from only a single node this symmetric difference is identical to the union of pstar(g,n) over all such atomic sets n . Thus pstar(g,n) which is defined as the union of pstar(g,n_o) over all such n_o is identical to pcob(g,n). A similar chain of definitions leads to the identity of ncob and nstar. However, cob and star are distinct since cob is defined as the symmetric difference of pcob and ncob whereas star is defined as the union of pstar and nstar (this, of course, applies only to the case of directed graphs; in the undirected case, star and cob are defined directly).

As an illustration of the preceding, consider g as in Fig. 9:

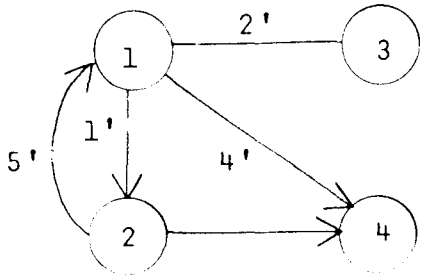

Fig. 9

Let n = {1,3}

pstar(g,n)=pstar(g,1) + pstar(g,3)={1',2',4'} + nl = {1',2',4'} = {1',2',4'} // nl = pcob(g,1)//pcob(g,3)= pcob(g,n)

nstar(g,n)=nstar(g,1) + nstar(g,3) = {5'} + {2'}= { 5',2' }= {5'} // {2'} = ncob(g,1)//ncob(g,3)

star(g,n)=pstar(g,n) + nstar(g,n) = {1',2',4'} + {5',2'} = { 1',2',4',5' }

but cob(g,n)= pcob(g,n) // ncob(g,n) = {1',2',4'} // {5',2'}

= {1',4',5'} ≠ star(g,n).

Although the SETL routines pstar and nstar above suffice for pcob and ncob, we present routines which mimic the theoretical definition directly:

```
definef pcob(g,n);
local p;
return if atom n then pstar(g,n)
                 else [/:p∈ n] pcob(g,n);
end pcob;
```

```
definef ncob(g,n);
local p;
return if atom n then nstar(g,n)
                 else [/:p∈n] ncob(g,n);
end ncob;
```

The operator pbd and nbd are defined by a similar process as pcob and ncob. Recall that bd(g,a), where a is a set of arcs, is the set of nodes incident to exactly one arc of a at only one point. For a directed graph, this same

set may also be defined as follows: first define the operators

pbd(g,a)≡pinc(g,a)

nbd(g,a)≡ninc(g,a)

if a is a single arc and then extend the definitions by symmetric differences for the case where a is an arc set. Note that unlike the case of pcob and ncob a node may be the starting or terminating point of more than one arc so that the symmetric difference is not identical to the union. For this reason, pbd and nbd operating on a set of arcs a, are distinct operators from pinc and ninc. Finally, the operator bd can be defined for a directed graph by

bd(g,a)=pbd(g,a)//nbd(g,a).

To illustrate these points, consider again, the graph of Fig. 9 and let a={4',2',3'} .

```
pinc(g,a)=pinc(g,4') + pinc(g,2') + pinc(g,3')
         ={1} + {1} + {2} = {1,2}
pbd(g,a) =pinc(g,4') // pinc(g,2') // pinc(g,3')
         ={1} // {1} // {2} ={2}
ninc(g,a)=ninc(g,4') + ninc(g,2') + ninc(g,3')
         ={4} + {3} + {4} = {3,4}
nbd(g,a) =ninc(g,4') // ninc(g,2') // ninc(g,3')
         ={4} // {3} // {4} ={3}
inc(g,a) =pinc(g,a) + ninc(g,a)
         ={1,2} + {3,4} = {1,2,3,4}
bd(g,a)=pbd(g,a) // nbd(g,a) = {2} // {3} = {2,3}
```

The SETL routines for pbd and nbd are:

```
definef pbd(g,a);
local p;
return if atom a then pinc(g,a)
                 else [/:p∈a] pbd(g,p);
end pbd;


definef nbd(g,a);
local p;
return if atom a then ninc(g,a)
                 else [/:p∈a] nbd(g,a);
end nbd;
```

Note that all of the above primitive routines are coded assuming the extended triple representation of graphs. However, by recoding them to apply to any of the other representations, one can specify higher-level algorithms directly to that representation. Alternatively, one could use the previously described conversion routines to convert back and forth between any other representation and extended triples.

IV. Graph algorithms in SETL:

The preceding conversion routines and fundamental routines can be used in SETL programs to produce more complex graph algorithms. Indeed, this is the philosophy behind GRAAL where these routines are provided as built-in functions. Thus, the developers of GRAAL also have developed a large set of routines in FGRAAL for such purposes as generating random graphs, finding induced graphs for a set of arcs or nodes, finding spanning trees of various forms and discovering connected components, cycles, cocycles, and blocks of graphs.

Unfortunately, these algorithms are rather complex and it is not at all apparent from the FGRAAL code what the algorithm purports to do and how it does it. Part of this problem is the lack of recursive routines in FORTRAN, and so in its extension FGRAAL. It is our belief that the development of set-theoretic recursive definitions of the algorithms would be trivially translatable into SETL and would produce cleaner code.

However, we did not undertake such analysis of the algorithms, but rather were interested in producing SETLB programs which produced working graph theoretic algorithms. The path of least resistance (although by no means most

creative and useful) was to, as far as possible, transcribe the FGRAAL algorithms into SETLB. The first step, of course, was to code the conversion and primitive routines. These have been (hopefully) completely debugged and accomplish their avowed aims. Once those routines were coded, they could be used in SETL algorithms which mimicked the FGRAAL algorithms given by the GRAAL developers. This, of course, uses only a fraction of the SETL power. For example, a common strategem in these algorithms is to trace down several paths and then retrace that path which is found to work. A more natural SETL approach would be to save the path that one is tracing along with the node that one is at in the trace so that when the "right" node is finally found, the path to it is immediately available.

Two peculiarities, one of SETL and one of GRAAL caused minor difficulty in the transcription:

1. Since all SETLB argument transmission is by value (except for sets and tuples, when what is actually passed is a pointer), in order to return a graph which has been created within a routine, the routine must be coded as a function if it is to be used as an external procedure (which is of course desirable in coding a set of general purpose algorithms). This also means that two distinct graphs must be returned as a pair. FGRAAL, being an extension of FORTRAN, passes arguments by value-result and so can return a result in one or more parameters.

2. The GRAAL phenomenon, mentioned earlier, of using union and set difference to include the SETLB WITH. and LESS. operators. This problem continually crops up since a single FGRAAL statement such as

    A=B.UN.C

    can have C be a set on one iteration of the loop and be a single node or arc on the second. A uniform way to avoid this would be never to use single arcs or nodes as values of SETLB variables, but always sets of a single element, e.g. to write

    x= {arb s}

    rather than x= arb s. Another peculiarity of GRAAL is that one can ask for the n^{th} element of a set by using the primitive ELT(N,S). Thus, to insure that A and B are two distinct arbitrary elements of S one would write:

    A= ELT(1,S)

    B= ELT(2,S)

    This would have to be encoded in SETL by

    A= arb S;

    B= arb (S less A);

    GRAAL also provides for what Knuth calls "deques" and which the GRAAL authors call "staques", i.e. linear lists in which items can be inserted or deleted at either end. Such staques and insertions and deletions from them can easily be simulated using SETL tuples.

REFERENCES

1. W. C. Rheinboldt, V. R. Basili, and C. K. Mesztenyi, *On a Programming Language for Graph Algorithms*, Bit 12 (1972), 220-241.

2. W. C. Rheinboldt, V. R. Basili, and C. K. Mesztenyi, *GRAAL- A Graph Algorithmic Language* , reprinted from: Sparse Matrices and Their Applications, edited by D. J. Rose and R. A. Willoughby. NSF Grant GJ-1067 and NASA Grant NGL-21-002-008, Computer Science Center, University of Maryland.

3. W. C. Rheinboldt, V. R. Basili, and C. K. Mesztenyi, *Graph Structure Algorithms in FGRAAL* , Technical Report TR-225, N00014-67-A-0239-0021 (NR-044-431), NGL-21-002-008, January 1973.

4. W. C. Rheinboldt, V. R. Basili, and C. K. Mesztenyi, *FGRAAL- Fortran Extended Graph Algorithmic Language*, Technical Report TR-179, NGL-21-002-008 and N00014-67-A-0239-0021 (NR-044-431), March 1972.