PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 11/72

G/PL/I-EXTENDING PL/I FOR GRAPH PROCESSING

by

C. S. Santos
and
A. L. Furtado

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

G/PL/I - EXTENDING PL/I FOR GRAPH PROCESSING

C. S. Santos
Assistant Professor
Universidade Federal do Rio Grande do Sul

and

A. L. Furtado
Associate Professor
Computer Science Department
PUC/RJ

This paper was published in the Fourth International Symposium on Computer and Information Sciences - Miami, Beach, Florida - December 14 - 16/ 1972

Series Editor: Prof. A. L. Furtado

December/ 1972

ABSTRACT

G/PL/I extends PL/I to handle both directed and undirected graphs, which may be multi - graphs. An arbitrary number of attribute-value pairs can be associated with the graph itself and with it nodes and edges.

Special kinds of sets together with set theoretic operations are provided.

The implementation uses pre-processing. It consists of a supervisor and modules for several extensions to PL/I, G/PL/I being one of them.

I. Introduction

G/PL/I extends PL/I to allow graph processing. Some of its features are:

1. Both directed and undirected graphs, which may be multigraphs, are handled; an arbitrary number of attribute - value pairs can be associated with the graph itself and with its nodes and edges.

2. A heterogeneous linked representation is used which facilitates the structural operations of creation, dele - tion, and interrogation of graph and graph elements.

3. Special kinds of sets involved in graph processing together with set - theoretic operations are provided.

4. Initialization, input/output, loop control, and several auxiliary processes are also included.

5. G/PL/I gives the user a concise and natural notation for writing graph algorithms, by extending the syntax of PL/I. Although the present implementation is intended for batch processing we claim that the notation is well suited for a conversational environment.

6. The implementation uses PL/I's pre-processor. It consists of a supervisor and modules for several extensions to PL/I, G/PL/I being one of these.

A comparison with other existing graph processing systems would show that G/PL/I takes from GRASPE [1] its creation, deletion , and interrogation operations, and that it follows GRAAL [2] and GEA [3] insofar as it extends a host language. The choice of PL/I as the host language was motivated by its growing use as a general purpose lan - guage, and by the presence of several very convenient features, notably a pre-processor in high level language, and based structures. For another system using PL/I see [4].

After a brief description of G/PL/I and its implementation, we present a program to determine the coefficients of the chromatic polynomial of an undirected graph, its chromatic number, and all of its optimal colorings, by using a version of Zykov's algorithm.

II. An Informal Description of the Extension

In describing G/PL/I we begin by introducing its notation in a very informal way.

Let G be a graph, and N1 and N2 any two nodes. In order to represent an oriented edge A from N1 to N2 we write:

```
A: N1 -> N2
```

We now consider the creation (+), deletion (-), and interrogation (?) operations, having graphs and graph elements (nodes or edges) as operands.

The value of the following expression would be G with edge A added to it:

G + A: N1 -> N2

addition of N1 and N2 being implied, if they are not already in G.

The removal of A, retaining N1 and N2, is indicated by

G - A: N1 -> N2

or implicitly by either of the node removal expressions

G - N1
G - N2

However, as one might expect,

G + N1
G + N2

merely indicate the addition of nodes N1 and N2 respectively, without implying the addition of an edge A.

If the edge is in G, and we write for instance:

G A: N1 -> ?
G ?: N1 -> N2
G ?: N1 -> ?

we get, respectively, the sets {N2} , {A} , {(A,N2)}.

When in an interrogation we do not care what a particular element (edge or node) is, an asterisk is used in the position of that element. Thus

G *: N1 -> ?

would again result in {N2}. For a more interesting example, suppose G has A1: N1 -> N2, A2: N1 -> N3, A3: N1 -> N4. Then

G *: N1 -> ?

would give {N2, N3, N4} , and here we stress the fact that the result of an interrogation is always rendered as a set.

Another case where sets of several elements are to be expected arises when a multi-graph is involved. If G has A1: N1 -> N2, A2: N1 -> N2, then

G ?: N1 -> N2

gives {A1, A2}.

Inpoiting (<-) and symmetric (<->) arrows are also permitted; the latter is the device used in representing undirected graphs.

The following expressions show, respectively, how to indicate the creation of an empty graph, the deletion of an entire graph, and the interrogations to obtain the set of all nodes, the set of all edges, and the set of all triplets (edges together with the two nodes incident to them):

```
G +
G -
G ?
G ?:
G ?:?
```

What we call composite expressions is a convenient way to indicate several additions and deletions in a single expression, such as

```
G ((-N1, + A2: N3 <- N4, - A7: N3 -> N8))
```

meaning the graph G plus the edge A2: N3 <- N4, and minus the node N1 and the edge A7: N3 -> N8.

There is a distinction between

```
G + N1
```

and

```
G + 'N1'
```

In the former case we are dealing with a character string variable whereas in the latter we have a character string constant which represents itself. Of course a variable will stand for a previously assigned constant, and the constant that is assigned to it immediately before the + operation is executed will be the actual element to be added to the graph.

The reference to a graph (G in the example) is a special kind of variable that must be declared and initialized as follows:

```
DECLARE  (G1, G7(3,4))  @GRAPH;
. . . . . .
@INIT((G1, G7));
```

noting that G7 is an array of graphs. Procedures having graphs as arguments and graph - valued procedures are permitted. Variables of the type graph have the same properties as pointer variables.

A graph is actually created (allocated) when an expression of the form

```
G  +
```

is encountered. The name of the graph is a character string constant identical to the variable referring to it, unless another name is specified as in

```
G  'GNAME'  +
```

Attribute - value pairs are adjoined to graphs, nodes, or edges by the operator / :

```
G  /  AT  = VL
G  N  /  AT = ((VL1,VL2,VL3))
G  A: /  AT = (VL)
```

noting that an edge is represented without specifying its endpoints, and that the value corresponding (=) to an attribute may be a list of values, indicated by two pairs of brackets. Each value can be a character string constant or any variable (whose value will be converted to a character string constant), or a set variable or set expression within single brackets.

The addition of pairs causes the removal of pairs with the same attribute that happen to be attached to the graph or graph element.

Deletion and interrogation operations are defined in a manner analogous to the one for graph elements. Thus

a.	b.
G / AT -	G / -
G N / AT -	G N / -
G A: / AT -	G A: / -

indicate the deletion of the pair with the given attribute (case a) or all attribute - value pairs (b), and likewise

a.	b.
G / AT = ?	G / ?
G N / AT = ?	G N / ?
G A: / AT = ?	G A: / ?

will deliver the set of values corresponding to the given attribute (case a) or the set of all attribute - value pairs (case b).

Several additions and deletions of attribute - value pairs are permitted in composite expressions:

G N / ((A1 = V1, A2 = ((V21, V22)), A3 -))

When an element (N or A) from a graph G1 is being added to another graph G2 one frequently wants to carry to G2 the attribute - value pairs that the element has in G1. This is indicated by the "qualification" operator (.):

```
G2 + G1 . N
G2 + G1 . A:
```

A graph expression can be assigned to a graph variable, as in the following statements:

```
G2 = G1 + N
G1 = G1 + A:N1 -> N2
```

Graph assignment statements and graph expressions are given to the preprocessor enclosed in brackets and preceded by the keyword @G. The same requirement is done for the input/output statements:

```
G <-
G ->
```

For input a graph should be represented as in the example below:

```
('GNAME'(AT1' = 'VL1', 'AT2' = 'VL2'),
 'N1'('ATN1' = 'VLN1'), 'A3':'N5' -> 'N9'('ATA3' = 'VLA3'))
```

the name of the graph coming first and then the nodes and edges in any order; both the graphs and its elements are optionally followed by their attribute - value pairs.

For output the graph in the example above would be printed in the form:

```
GRAPH
   GNAME                    (AT1 = VL1, AT2 = VL2)
NODES
   N1                       (ATN1 = VLN1)
   N5                       ( )
   N9                       ( )
EDGES
   A3:N5 -> N9              (ATA3 = VLA3)
```

A number of auxiliary processes are provided as G/PL/I procedures. They allow the conversion of the standard representation in G/PL/I (linked structure) into and from adjacency matrix and incidence matrix representation; testing if an edge is directed; conversion of the standard representation into and from a character string representation suitable for saving a graph in secondary storage.

Three types of sets are provided. They were introduced exclusively as an auxiliary facility.

Type one sets can be sets of nodes, of edges (the names of the edges only), or of values. Types two and three are sets of edges, type two giving the edge name and one of its endpoints (sets of ordered pairs), and type three the edge name and both its endpoints (set of triples). Type two sets are also used as sets of attribute - value pairs.

As in the case of graphs sets must be declared and initialized:

```
DECLARE (S1, S2(2, 20)) @SET;
. . . . . .
@INIT ((S1, S2));
```

Operations on sets and comparison between sets are provided, the chosen notation being:

|	union
ε	intersection
ε¬	difference
+	adding an element
-	deleting an element
<=>	equality (mutual inclusion)
>=	containment
>	proper containment
<=	inclusion
<	proper inclusion
¬=	unequality

Set expressions are evaluated from left to right, and since no operator priority is assumed brackets may be required to ensure the desired order of evaluation. A set variable represents an empty set whenever its value is NULL.

Set assignment is also included. As for graphs, set assignments must be enclosed in brackets preceded by a keyword, which in this case is @S.

If a set is created by a set expression and is not assigned to a set variable it is regarded as a temporary set and as such it is freed after the statement in which it arises has been executed.

Loop control is achieved by means of a special statement, as in:

```
@DO EACH (X IN S);
- - - - - - - - - -
    statements
- - - - - - - - - -
@END;
```

where the character string variable X takes in turn the value of each element from type one set S. In case S is a type two or three set, X will also be a set variable and it must be enclosed in brackets in the @DO statement.

Procedures are provided for converting a character string into a set element; retrieving the first or the $i^{\underline{th}}$ element of a set ; determining the position (index) of a given element in a set - it is important to note that @INDEX(S,X) > 0 means X ε S; determining the type of a set; computing the cardinality of a set; converting the standard set representation into or from a representation suitable for saving sets in secondary storage. Here we should note that since we are dealing with unordered sets, the term position merely refers to the order imposed by the representation.

The example at the end of the chapter will hopefully illustrate how graph processing benefits from this limited set facility.

III. Implementation Considerations

The linked information structures used to implement graphs and sets are modular, in the sense that each part (module) will be included only when required to record an effectively supplied information.

The modules and their composition are listed below. Note in particular that multi-graphs are representable.

1. Active graph (entry in the graph directory)
   - graph name
   - address of the header of the graph
   - address of the next entry in the directory

2. Graph header
   - address of the list of attribute-values of the graph
   - address of the list of nodes of the graph
   - address of the list of edges of the graph

3. Node
   - node name
   - address of the list of edges incident to the node
   - address of the list of attribute-values of the node
   - address of the next node of the graph

4. Edge
   - edge name
   - tag that indicates whether or not the edge is directed
   - address of the initial node of the edge
   - address of the terminal node of the edge (of course if the edge is undirected the terms initial and terminal are equally applicable to both endpoints)
   - address of the list of attribute-values of the edge
   - address of the next edge of the graph

5. Link from node to edge
   - address of the edge incident to the node
   - address of the next link (leading to another edge incident to the same node)

6. Attribute-value pair
   - attribute name
   - value
   - address of the next attribute-value pair of the graph or graph element

7. Entry in the set directory
   - tag indicating the status of the set (temporary or permanent)
   - address of the header of the set
   - address of the next entry in the directory

8. Set header
   - set type (1,2, or 3)
   - address of the first element in the set

9. Set element
   - element name
   - address of the next element in the set (here the terms first and next refer only to the order imposed by representation)

As previously indicated graph structures are explicitly allocatted and deallocated; since no garbage collection mechanism is provided in the present implementation, the user should take the burden of deallocating the graphs as they are no longer needed.

Both nodes and edges are "local" to each graph. This is at variance with systems like GRASPE [1] , where nodes are global and only the edges are local.

Another relevant point about the implementation is the use of the standard IBM pre-processor to convert G/PL/I programs into valid PL/I programs.

Since ambiguities are avoided a one - pass scan through graph or set expressions is sufficient to determine what actions should be taken. Accordingly, calls are generated to one or more procedures , declarations, and other statements.

The IBM pre-processor is a high level (PL/I-like) language, which makes it relatively easy to write, understand, and change (possibly expand) anything in the presently implemented version. Moreover - and this is true for any pre-processor implementation - the users of the non - extended language are not affected, since the compiler has not been changed.

We are also using the IBM pre-processor for several other extensions to PL/I (for list processing, additional control statements, pattern-matching, etc.). The extension modules operate under a supervisor, involving catalogued procedures and some simple utility programs; a single JCL card indicates what extension modules will be combined in a particular program.

IV. An Example

In order to illustrate some of the features of the language we include a program (see Appendix) to compute the coefficients of the chromatic polynomial of an undirected graph and all its optimal colorings (to within renaming the colors) by a version of Zykov's algorithm.

The chromatic polynomial is given by a recursive function Z on G = a graph -, n - the number of nodes of G -, and t - any indeterminate - (see [5] page 145):

```
Z(G,n,t) = if complete (G,n) then f(n) /* basis */
           else Z(G1,n,t) ⊕ Z(G2,n-1,t) /* recursion step */
```

where:

f computes the coefficients of the chromatic polynomial of a complete graph G by expanding $t(t-1)\ldots(t-n+1)$;

G1 is obtained from G by adding an edge linking two arbitrarily selected non-adjacent nodes u and v;

G2 is obtained from G by fusing the same non-adjacent nodes u and v;

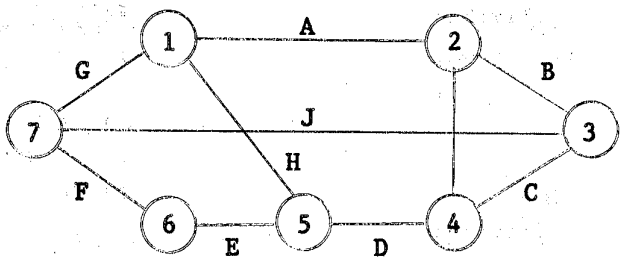
⊕ adds the coefficients of corresponding terms of the chromatic polynomials of G1 and G2.

It has been shown [6] that the complete graphs with the smallest number of nodes represent the optimal colorings, each node resulting from fusing those of the nodes from the original graph to be assigned the same color. As a consequence the number of nodes of such complete graphs is the chromatic number of the original graph (which is also the least positive integer value of t yielding a non-zero value for the chromatic polynomial).

The exponencial nature of the algorithm requires the introduction of suitable branch and bound criteria. No such thing was done here, since our purpose is to show briefly how G/PL/I is used.

An interesting feature of the program is the use of the attribute - value lists of the nodes of the complete graphs for storing the set of original nodes that were fused to form them.

As an example, consider the graph:



whose input representation is:

```
('INPUT', 'A':'1' <-> '2', 'B':'2' <-> '3', 'C':'3' <-> '4',
'D':'4' <-> '5', 'E':'5' <-> '6', 'F':'6' <-> '7', 'G':'1' <-> '7',
'H':'1' <-> '5',  'I':'2' <-> '4',  'J':'3' <->'7')
```

The output consists of the input graph itself, the complete graphs generated by the process (the set of attribute - value lists of their nodes exhibiting the optimal colorings), and the coefficients of the chromatic polynomial.

The chromatic number of the given graph is 3, and the following optimal colorings were obtained:

{{1,3,6} , {4} , {7,5,2}}
{{7,5,2} , {4,6} , {1,3}}
{{7,4} , {2,5} , {1,3,6}}
{{7,5,2} , {3} , {1,4,6}}
{{1,4} , {6,3} , {7,5,2}}
{{1,4,6} , {5,3} , {7,2}}

the coefficients of the chromatic polynomial being:

(1, -10, 44, -109, 159, -127, 42)

V. Directions for Further Developments

Our first objective was to develop a good notation for writing graph algorithms. Time and core storage requirements optimization will become increasingly important issues as improved versions of G/PL/I and the other extensions are implemented, leading possibly to their formal incorporation in an extended language definition and to the design of a new and complete compiler.

Adaptation to a conversational environment is also being considered. The use of display units would add the ability to visualize graphs in the usual way (see [7,8] for a very interesting example of this).

A simpler and yet effective alternative for adding power to the system is to write as library procedures certain algorithms, which other more complex ones will utilize as sub-algorithms. This approach was used in [9]. Among such sub-algorithms we are including automorphism partitioning [10], palm tree construction [11], etc.

Also, since it is not clear that any particular graph representation (linked lists, adjacency structure, adjacency matrix, etc.) is the best in all respects for all algorithms, mappings between the preferred representation and the other most usual representations are desirable. We have provided some of these mappings as auxiliary procedures (see section II), and others will probably be added.

APPENDIX

```
CHROM:PROCEDURE OPTIONS(MAIN);
      DCL G @GRAPH,
          N BIN FIXED;
     @INIT((G));
     /* READS A GRAPH FROM CARDS AND PRINTS IT */
     @G(G <-);
     @G(G->);
     /* GETS THE NUMBER OF NODES */
     N=@SCARD(@G(G ?));
     BEGIN;
          DCL CPC(N) DEC FIXED(7,0);
          /* CALLS   Z */
          CALL Z(G,CPC,N);
          /* PRINTS THE COEFFICIENTS OF THE */
          /* CHROMATIC POLYNOMIAL           */
          PUT PAGE LIST('CHROMATIC POLYNOMIAL COEFFICIENTS ');
          PUT SKIP EDIT(CPC)((N) F(8));
     END;
     Z:PROCEDURE(G,CPC,N);
          DCL G @GRAPH,
              COLOR @GRAPH CTL,
              (S,S1) @SET,
              N BIN FIXED,
              X CHAR(12),
              I DEC FIXED INITIAL(0),
              (TN,CPC(*),AUX(N)) DEC FIXED(7,0);
          /* TN IS USED TO GENERATE THE NAME OF NEW EDGES, AND  */
          /* I IS USED TO GENERATE THE NAME OF THE COMPLETE     */
          /* GRAPHS TO BE STACKED                               */
          @INIT((S,S1));
          TN=10000;
          CPC,AUX=0;
          /* PUTS THE NAME OF EACH NODE IN ITS */
          /* OWN ATTRIBUTE-VALUE LIST          */
          @DO EACH(X IN @G(G ?));
               @G(G=G X/'N'=X);
          @END;
          /* CALLS   Z1 */
          CALL Z1(G,N);
          /* PRINTS ALL MINIMUM COMPLETE GRAPHS */
          DO WHILE(ALLOCATION(COLOR));
               @G(COLOR ->);
               FREE COLOR;
               END;
          Z1:PROCEDURE(G,N) RECURSIVE;
               DCL (G,G1) @GRAPH,
                   (N,N1,K) BIN FIXED,
                   (ND1,ND2,ND3,EG1,GNAME) CHAR(12);
               @INIT((G1));
```

```
/* IF G IS COMPLETE THEN DETERMINE  */
/* ITS CHROMATIC POLYNOMIAL         */
IF (N*(N-1)/2) = @SCARD(@G(G ?:))
THEN DO;
     CALL F(N);
     IF ¬ALLOCATION(COLOR)
     THEN K=10000;
     ELSE K=@SCARD(@G(COLOR ?));
     /* IF THE NUMBER OF NODES IN G IS LESS THAN */
     /* THE NUMBER OF NODES OF THE COMPLETE      */
     /* GRAPHS IN THE STACK THEN EMPTY THE STACK */
     IF N < K
     THEN DO;
          I=0;
          DO WHILE(ALLOCATION(COLOR));
               @G(COLOR=COLOR -);
               FREE COLOR;
          END;
          END;
     /* IF THE NUMBER OF NODES IN G IS LESS OR   */
     /* EQUAL TO THE NUMBER OF NODES OF THE      */
     /* COMPLETE GRAPHS REMOVED FROM OR IN THE   */
     /* THE STACK THEN ADD G TO THE STACK        */
     IF N <= K
     THEN DO;
          ALLOCATE COLOR;
          @INIT((COLOR));
          I=I+1;
          GNAME='COLOR'||SUBSTR(I,5,4);
          @G(COLOR=COLOR GNAME +);
          @G(COLOR=G);
          END;
     END;
/* G IS NOT A COMPLETE GRAPH */
ELSE DO;
     /* CALLS NOTADJ TO GET TWO NON-ADJACENT     */
     /* NODES  ND1  AND  ND2                     */
     CALL NOTADJ(G,ND1,ND2);
     TN=TN+1;
     EG1='@E'||SUBSTR(TN,9,2);
     /* CREATES G1 FROM G BY ADDING AN EDGE      */
     /* LINKING  ND1  AND  ND2                   */
     @G(G1=G+EG1:ND1<->ND2);
     /* CALLS  Z1  RECURSIVELY                   */
     CALL Z1(G1,N);
     /* CREATES G1 FROM G  BY FUSING ND2 TO ND1. */
     /* FOR THIS ND2 IS DELETED BUT THE          */
     /* ATTRIBUTE-VALUE LIST OF ND1 BECOMES THE  */
     /* UNION OF WHAT IT HAD BEFORE WITH THE     */
     /* LIST OF ND2, AND THE EDGES LINKING ND2   */
```

```
          /* WITH NODES NOT LINKED TO ND1 NOW BECOME   */
          /* INCIDENT TO ND1                           */
          @G(G1=G-ND2);
          @S(S1=@G(G ND1/'N'=?) | @G(G ND2/'N'=?));
          @G(G1=G1 ND1/'N'=(S1));
          @S(S=@G(G *:ND2<->?) &¬ @G(G *:ND1<->?));
          @DO EACH(ND3 IN S);
               TN=TN+1;
               EG1='@E'||SUBSTR(TN,9,2);
               @G(G1=G1+EG1:ND1<->ND3);
          @END;
          N1=N-1;
          /* CALLS Z1 RECURSIVELY. SINCE ND1 AND ND2   */
          /* ARE FUSED G1 HAS  N-1  NODES              */
          CALL Z1(G1,N1);
          /* G1  IS  DE-ALLOCATED                      */
          @G(G1=G1-);
          END;
END Z1;
NOTADJ:PROCEDURE(G,ND1,ND2);
     /* SEARCHES FOR ANY TWO NODES ND1 AND ND2 WITH   */
     /* NO EDGE LINKING THEM                          */
     DCL G @GRAPH,
         (ND1,ND2) CHAR(12);
     @DO EACH(ND1 IN @G(G ?));
          /* CHECKS THE DEGREE OF ND1 */
          IF @SCARD(@G(G *:ND1 <-> ?)) < N - 1
          THEN @DO EACH(ND2 IN @G(G ?));
                    IF ND1 ¬= ND2
                    THEN IF @G(G ?:ND1 <-> ND2) = NULL
                         THEN RETURN;
               @END;
     @END;
END NOTADJ;
 F:PROCEDURE(P);
     /* OBTAINS THE COEFFICIENTS OF THE CHROMATIC     */
     /* POLYNOMIAL OF A COMPLETE GRAPH WITH P NODES   */
     /* BY EXPANDING  P(P-1)(P-2)...(P-P+1)           */
     DCL (P,I,J) BIN FIXED;
     AUX(1)=1;
     IF P > 1
     THEN DO;
          AUX(2)=-1;
          DO I=2 TO (P-1);
               AUX(I+1)=-I*AUX(I);
               DO J=I TO 2 BY -1;
                    AUX(J)=AUX(J)-I*AUX(J-1);
               END;
          END;
          END;
```

```
                /* ADDS CORRESPONDINGLY THE COEFFICIENTS OF THE   */
                /* POLYNOMIAL TO  CPC, WHICH AT THE END WILL      */
                /* CONTAIN THE COEFFICIENTS OF THE  CHROMATIC     */
                /* POLYNOMIAL                                     */
                DO I=1 TO P;
                     CPC(I+N-P)=CPC(I+N-P)+AUX(I);
                     AUX(I)=0;
                END;
           END F;
      END Z;
END CHROM;
```

ACKNOWLEDGEMENT

We wish to thank our colleagues L. Kerschberg and Sueli Santos who carefully reviewed the manuscript of this paper.

REFERENCES

1. Pratt, T. and Friedman, D., "A Language Extension for Graph Processing and its Formal Semantics". Comm. ACM 14, 460-467 (1971)

2. Rheinboldt, W. et al, "On a Programming Language for Graph Algorithms." Tech. Rep. TR-158. University of Maryland, 1971.

3. Crespi-Reghizzi, S. and Morpurgo, S., "A Language for Treating Graphs." Comm. ACM 13, 319-323 (1970).

4. Chase, S., "Analysis of Algorithms for finding all spanning trees of a graph." Tech. Rep. 401. University of Illinois, 1970.

5. Harary, F., "Graph Theory." Addison-Wesley, 1969.

6. Corneil, D. et al. Private communication.

7. Christensen, C., "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language." In "Interactive Systems for Experimental Applied Mathematics." (Klerer and Reinfelds, eds.). Academic Press, 1969.

8. Christensen, C., "An Introduction to AMBIT/L, a Diagrammatic Language for List Processing." Proc. Second Symposium on Symbolic and Algebraic Manipulation, 1971.

9. King, C., "A Graph Theoretic Programming Language." Ph.D. Dissertation, University of West Indies, 1970.

10. Corneil, D., "An Algorithm for Determining the Automorphism Partitioning of an Undirected Graph." Working Paper, University of Toronto.

11. Tarjan, R., "Depth-First Search and Linear Algorithms." Working Paper, Stanford University.