Graph Programming Language One (GPL/I)

Kazuyuki Imafuji*

ABSTRUCT

This paper describes a language for handling graphs as a new type of data. This language is called Graph Programming Language One(GPL/I) and is defined as an extension of PL/I. It consists of two kinds of statements, that is, GPL/I and PL/I. One GPL/I statement is translated into PL/I statements by means of a pre-compiler(GPL/I compiler) which was developed for IBM System 370/165. In this compiler, graphs are represented by one-directional lists and new graphical operations are defined.

A sample program to get one spanning tree of a given graph is also shown.

1. INTRODUCTION

Graphs (linear graph) have been used in various fields as a useful tool to solve problems, such as electrical circuit analysis, network flow problems, PERT diagrams in linear programming etc.

When these problems are solved by means of graphs, connection matrices are usually used(1) and are handled by computer. But it requires much labor to construct a connection matrix as an input datum to the computer. And if operations on a graph or between graphs are performed, then various subprograms are required. So, graph users usually develope these subprograms individually. In order to emancipate users from these troubles, some languages dealing with graphs have been designed and developed by many investigators. R. C. Read et al. has designed Graph Theory Programming Language(GTPL)(2). It takes an existing language, FORTRAN, as its basis and adopts the extra statements such as defi-

This paper first appeared in Japanese in Joho－Shori (Journal of the Information Processing Society of Japan), Vol. 13, No. 12 (1972), pp. 818～824.

* Mitsubishi Electric Corp.

nitions of graphs and grammatical structures necessary to handle graphs. On the other hand, S. Crespi-Reghizzi et al. has reported Graph Extended ALGOL(GEA) (3). From the standpoint of applications, associated functions concerning points and lines are adopted in GEA. As an extension of LISP 1.5, GRASPE1.5 has also been reported(4)(5). This was designed to solve pure graphical problems.

GPL/I language, GPL/I compiler and GPL/I system libraries constitute the GPL/I system. This system is based on PL/I language, because the language has new functions such as dynamic allocations of memories, structured data, and, especially, a list processing functions.

The language structure of GPL/I, the internal structure of graphs and an example of its usage are described in this paper.

For definitions of terms on graphs, the reader may refer to Berge(6) or Harary (7).

2. STRUCTURE OF GRAPH STATEMENTS

GPL/I language is an extension of PL/I language and its syntax is generally dependent on the syntax of PL/I. The statements for handling graphs are called GPL/I statements and are distinguished from PL/I statements. Only GPL/I statements will be described here. The reader may refer to the manual on PL/I statements(8).

2.1. Declaration of graphs

The user of GPL/I may handle simple graphs, multi graphs, simple digraphs and multi digraphs. Fig. 1 shows graph types and their examples.

The graph type which is called attribute must be declared in a DCL statement.

```
$DCL NET1 SIMPLE GRAPH,
     NET2 MULTI DIGRAPH;
```

By this DCL statement, NET1 and NET2 are registered as a simple graph and a multi digraph, respectively.

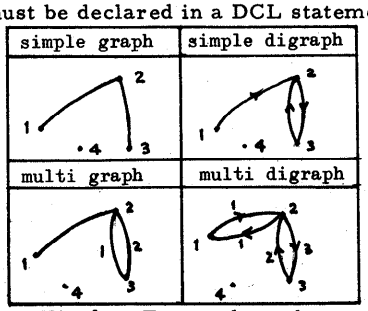

Fig. 1. Types of graphs

Points and lines of a graph may be

refered by following forms.

graph name(EE), (1)

graph name(EE1, EE2[, EE3]), (2)

where (1) is used for referring a point and (2) is for a line. EE is an abbreviation of the "element expression". EE3 in (2) may be used in case of a multi graph. In case of a directed graph, EE1 and EE2 in (2) are an ordered pair, but in case of a simple graph their order is meaningless.

Examples of the forms (1) and (2) are shown below.

NET(2) a point of point number 2 in the graph NET,

NET(I, J+1) a line connecting two points whose numbers are the value I and J+1 in the graph NET.

In many graphical problems, it is necessary to assign functions over the set of points or the set of lines. Users of these functions which are called a point function or a line function must declare them in DCL statements.

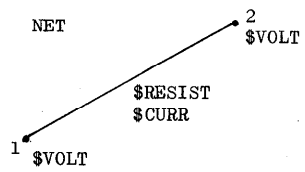

Fig. 2. Example of point & line functions

A point function and line functions in Fig. 2 must be declared as follows.

```
$DCL 1 NET SIMPLE GRAPH, 2 $VOLT PFUNC,
        2 $RESIST LFUNC, 2 $CURR LFUNC;
```

The following statements show some examples of the usage of these functions.

```
$$VOLT(NET(1))=20.0;    $$RESIST(NET(1,2))=5.0;
```

2.2. Operations on graphs

Four kinds of operations between graphs are defined: a complement graph of a given graph; a sum of graphs; a difference of graphs; and an intersection of graphs.

(1) Complement of a graph

This operation is a unary operation and its operator is denoted by '¬'. A comple-ment graph ¬G of a given graph G consists of the complete graph made up of all points in G except the lines which are contained in G and the isolated points produced as a result of deleting the lines contained in G from the complete graph.

(2) Sum of graphs

This operation is a binary operation and its operator is denoted by '+'. A sum graph G1+G2 of two given graphs G1 and G2 consists of points and lines which are contained in G1 or G2, or in both.

(3) Difference of graphs

This operation is also a binary operation and its operator is denoted by '-'. A difference graph G1-G2 of two given graphs G1 and G2 consists of points in G1 except isolated points, which are produced as a result of deleting the lines contained in G2 from G1, and lines in G1 not contained in G2. G1 must completely contain G2, that is, G2 must be a subgraph of G1.

(4) Intersection of graphs

This operation is a binary operation and its operator is denoted by '*'. An intersection graph G1*G2 of two given graphs G1 and G2 consists of points and lines which are contained in both G1 and G2, but isolated points which are produced as a result of this operation are excluded.

As operations carried out between elements contained in a graph, the following are defined; addition and deletion of points or of lines, change and exchange of names of points.

2.3. Other GPL/I statements

The following is the outlines of new statements.

(1) Assignment statement

The graph on the right hand side of the equality sign or the result of graphical operations on the same side (whose attributes are converted into those of the graphs on the left hand side, if necessary) is assigned to the graphs on the left hand side.

(2) IF statements

This statement tests the existence of a specified element of the given graph or the specified character of the given graph and controls the flow of the executions according to the result of the test. GPL/I has two types of the IF statement.

$IF {¬} specified element THEN unit1 ELSE unit2 (3)

$IF graph name [¬]= IF keyword THEN unit1 ELSE unit2 (4)

In case of (3), if a graph contains a specified element, "unit1" is executed and then control passes to the statement following the IF statement; otherwise, "unit2" is executed and then control passes to the next statement. In case of (4), if a graph has a specified character determined by an IF keyword, "unit1" is executed; otherwise, "unit2" is executed. The executions following the execution of "unit1" or "unit2" are the same as those in (3). IF keywords are listed in Table 1.

Table 1.

List of IF keywords

Keyword	Meaning
COMPL	Is graph complete or not?
CONT	Is graph connected or not?
CYCL	Is cycle contained or not?
EMPT	Is graph empty or not?
FORST	Is graph forest or not?
PATH(I,J)	Is path from I to J contained or not?
REGR	Is graph regular or not?
REG(N)	Is graph n-regular or not?
TREE	Is graph tree or not?

(3) DO statement

The DO statement delimits the start of DO-group and provides for controlled repetitive execution of all statements in that DO- group on all points or lines in the specified graph. This statement is used with $END statement.

(4) Built-in function

Many kinds of built-in functions on graphs are listed in Table 2.

Table 2.

List of Functions

Functions	Meaning
I@COMP(G)	Number of components
I@DEG(G,K)	Degree of point K(graph)
I@DEGI(G,K)	Indegree of point K(digraph)
I@DEGO(G,K)	Outdegree of point K(digraph)
I@EDGE(G,K)	Neighbour point of point K
I@TYPE(G)	Type of graph
I@LINE(G)	Number of lines
I@MAXD(G)	Maximum degree(graph)
I@MAXDI(G)	Maximum indegree(digraph)
I@MAXDO(G)	Maximum outdegree(digraph)
I@MAXN(G)	Maximum point number
I@MIND(G)	Minimum degree(graph)
I@MINDI(G)	Minimum indegree(digraph)
I@MINDO(G)	Minimum outdegree(digraph)
I@MINN(G)	Minimum point number
I@NODE(G)	Any point number
I@POINT(G)	Number of point

3. GPL/I COMPILER AND LIBRARIES

3.1. Internal representation of graphs

Although graphs are usually processed in the form of connection matrices in the computer, following two problems typically arise:

(1) The efficiency of utilizing core memories becomes lower as the size of a graph grows larger. For example, the connection matrix used in power system contains about 4×10^6 elements

but only about 3×10^3 elements are meaningful (9).

(2) Matrices are usually represented in the computer by one dimensional arrays to get high efficiency of utilizing memories. But the operations of a graph such as additions or deletions of points or lines decrease the speed of executions.

In view of these facts, graphs are represented by one -directional lists.

Graph directory

NAME	PLIST	TYPE	POINT	LINE	PFUNC	LFUNC1	LFUNC2

Point list

NEXT	OTLP	INLP	NAME	OTDEG	INDEG	PFUNC

Line list

NEXT	NAME	LFUNC1	LFUNC2

Fig. 3. List structure

These lists are automatically treated by GPL/I, so that programmers need not be aware of the internal representation of graphs. As shown in Fig. 3, graph directories, point lists and line lists, constitute these lists. Fig. 4 shows the internal representation of the given graph. For simplicity, the point function and the line functions are omitted in this figure.

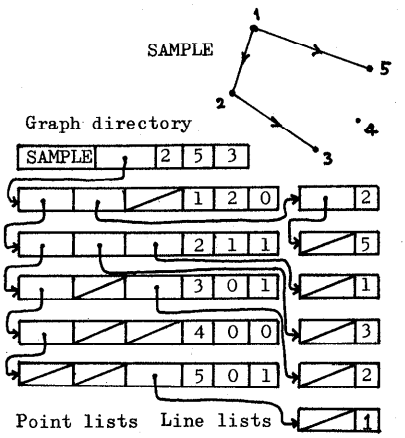

Fig. 4. Example of list staructure

3.2. Compiler

GPL/I source programs consist of GPL/I statements and PL/I statements. GPL/I compiler reads in the source programs and writes out PL/I statements without any change and converts a GPL/I statement to PL/I statements. GPL/I statements are led by the character '$' to be distinguished from PL/I statements. This simplifies the compiler. This GPL/I compiler is written in PL/I language.

3.3. System libraries

GPL/I system libraries are used to execute object programs of GPL/I compiler. These libraries contain about 60 subprograms.

4. AN EXAMPLE OF GPL/I PROGRAM

This program generates one spanning tree of a given graph A. A spanning tree B of a graph A is a subgraph of A which is a tree and contains all the points of A. The algorithm of this program is as follows.

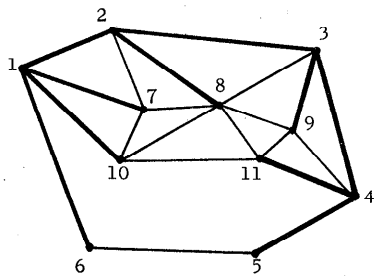

Fig. 5. Input graph & spanning tree

```
STMT NO.        GPL/I  SOURCE  STATEMENT          DATE = 72/04/15      PROC NO.= 1      PAGE = 1

      /* GPL/I PROGRAM EXAMPLE      'SPANNING TREE SUBPROGRAM'          */ SPS01000
  1   $MAIN    : PROC OPTIONS(MAIN);                                      SPS01100
        /* */                                                            SPS01150
  2   $          DCL (NET,TREE) GRAPH;                                   SPS01200
  3   $          GET NET;                                                SPS01250
  4              CALL SPANTRE(NET,TREE);                                 SPS01350
  5   $          PUT TREE;                                               SPS01400
  6   $          END MAIN;                                               SPS01450
```

```
STMT NO.        GPL/I  SOURCE  STATEMENT          DATE = 72/04/15      PROC NO.= 2      PAGE = 1

      * PROCESS('NOOPLIST,NOSTMT,NOATR,NOXREF');
                                                                  /* */ SPS01600
  1   $SPANTRE : PROC($A,$B);                                           SPS01650
        /* */                                                           SPS01700
  2   $          DCL (A,B,EMPT) GRAPH;                                  SPS01750
  3              DCL POINT BIN FIXED;                                   SPS01800
  4   $          IF A ¬= CONT THEN GO TO NOSPAN;                        SPS01850
  5              POINT=I@POINT(A)-1;                                    SPS01900
  6   $          DO A(I,J);                                             SPS01950
  7   $            B=B+B(I,J);                                          SPS02000
  8   $            IF B=CYCL THEN $ B=B-B(I,J);                         SPS02050
  9                IF I@LINE(B)=POINT THEN $ RETURN;                    SPS02100
 10   $          END;                                                   SPS02150
 11   NOSPAN   : PUT  EDIT  ('NO SPANNING TREE') (SKIP(1),COL(15),A);    SPS02200
 12   $          B=EMPT;                                                SPS02250
 13   $          END SPANTRE;                                           SPS02300
```

Fig. 6. GPL/I source list

Step 1. Choose any line u of A and delete u from A.

Step 2. Add u to B and look at the graph B. If B has a cycle, then delete u from B.

```
*** LIST OF GRAPH ***
     GRAPH-NAME = TREE       GRAPH-TYPE = SIMPLE GRAPH
          POINT NUMBER =  11
           1   2   7  10   6   8   3   9   4   5  11
          LINE  NUMBER =  10
           1 - 2     1 - 7     1 - 10     1 - 6     2 - 8     2 - 3
           3 - 9     3 - 4     4 - 5      4 - 11
```

Fig. 7. Output format for graph(spanning tree)

Step 3. If B now contains P-1 lines (where P is the number of points contained in the original A), the algorithm terminates. Otherwise, go back to Step 1.

The source program and the result on graph A shown in Fig. 5 are presented in

Fig. 6 and 7, respectively.

5. CONCLUSION

The language described in this paper is designed for simple operation on graphs (simple, multiple, nondirected and directed) such as complement, sum, difference and intersection between graphs and addition or deletion of points or lines etc. The followings are the characteristics of this language.

(1) It is possible to handle graphs as one type of data.

(2) PL/I compensates the defect of GPL/I because this contains PL/I language completely.

(3) The function of dynamic allocation of memories and the internal representation of graphs by one-directional lists reduce the size of region of memories and increase the speed of execution of programs.

(4) A program written in this language is shortened to three quarters compared with a program written in PL/I.

Therefore, this language may be considered a very useful tool to solve graphical problems.

ACKNOWLEDGMENT

I wish to thank Dr. J. Baba of Mitsubishi Electric Corp. for many helpful suggestions during the course of this work.

REFERENCES

(1) B. G. Busacker, T. L. Saaty: Finite Graphs and Networks; An introduction with applications, McGraw-Hill, New York, 1965.

(2) R. C. Read et al.: The Application of Digital Computer Techniques to the Study of Graph-Theoretical and Related Combinatorial Problems, Scientific Report No. UWI/CC 12, University of West Indies, 1969.

(3) S. Crespi-Reghizzi et al.: A Language for Treating Graphs, Comm. A. C. M. Vol. 13 No. 5, May 1970, 319-323.

(4) T. W. Pratt et al.: A Language Extension for Graph Processing and Its Formal Semantics, Comm. A. C. M. Vol. 14 No. 7, July 1971, 460-467

(5) D. P. Friedman et al.: GRASPE1. 5, A Graph Processor and Its Applications, Report RS1-69, University of Houston, August 1969.

(6) C. Berge: The Theory of Graphs and Applications, John Wiley & Sons Inc., New York, 1966.

(7) F. Harary: Graph Theory, Addison Wesly, Massachusetts, 1969.

(8) I. B. M. : IBM System/360 PL/I Reference Manual, C 28-8201-1

(9) E. C. Ogbuobiri et al.: Sparsity Directed Decomposition for Gaussian Elimination on Matrices, I. E. E. E. Trans. Power Appr. Sys. Vol. 1 PAS_89, No. 1, January 1970, 141.