NOTAS TÉCNICAS

VOLUME IV

Nº 1

GRAPH-PROCESSING WITH LISP

por

Adalberto S. Pfeffer, Adilson T. Medeiros, Mario T. Ribeiro

e Antonio L. Furtado

Rio de Janeiro, GB, Setembro de 1971

CENTRO BRASILEIRO DE PESQUISAS FÍSICAS

Av. Wenceslau Braz, 71

RIO DE JANEIRO

1971

ABSTRACT

Graph-processing languages or packages usually represent graphs under matrix or list form.

Although algorithms using lists are not in general the most efficient, they are easy to write and to understand.

New algorithms could first be sketched in LISP and then translated into another form, having in mind run time and core storage minimization.

In this tutorial paper a simple LISP representation for graphs in presented together with a number of basic functions.

All programs were tested on an IBM - 1620 of the Centro Brasileiro de Pesquisas Físicas.

1. FUNDAMENTAL ELEMENTS FROM LISP 1.5

This section is only included for the sake of completeness. The functions and expressions given below are those referenced from section 2.

a). Primitive Functions

cons [a; b] - builds a list consisting of its arguments.

car [a] - gives the first part of its composite argument.

cdr [a] - gives the second part of its composite argument.

eq [a; b] - a predicate, to test for the equality of two atomic symbols.

atom [a] - a predicate, to test if its argument is an atomic symbol.

b). Conditional Expressions

A conditional expression has the form:

$$[p_1 \rightarrow e_1; p_2 \rightarrow e_2; \ldots ; p_n \rightarrow e_n]$$

where each p_i is an expression whose value may be truth or falsity, and each e_i is any expression. The value of the conditional expression is the first (from left to right) e_i whose corresponding p_i is true.

c). Usual Auxiliary Functions

```
null [x] = [eq[ x; NIL ] → T;
            T → F]

equal [x; y] = [atom [x] → [atom[y] → eq[x; y];
                T → F];
                equal[car[x] ; car[y]] →
                        equal[cdr[x]; cdr[y]];
                T → F]

subst[x;y;z] = [equal[y;z] → x;atom[z] → z;
                T → cons[subst[x;y;car[z]];
                        subst[x;y;cdr[z]]]]

append[x;y] = [null[x] →y;
               T → cons[car[x];append[cdr[x];y]]]

member[x;y] = [null[y] → F;
               equal[x;car[y]] → T;
               T → member[x;cdr[y]]]

pairlis[x;y;a] = [null[x] → a ;
T → cons[cons[car [x]; car[y]];
        pairlis[cdr[x]; cdr[y]; a]]]
```

```
assoc[x;a] = [equal[caar[a];x] → car[a];
              T → assoc [x;cdr[a]]]

sublis[a;y] = [atom[y] → sub 2[a;y];
               T → cons[sublis[a;car[y]];
                        sublis[a;cdr[y]]]]

sub 2 [a;z] = [null[a] → z;
               eq[caar[a];z] → cdar[a];
               T → sub 2[cdr[a];z]]

add 1 [x] = x + 1

sub 1 [x] = x - 1

caar  [x] = car[car[x]]

cadr  [x] = car[cdr[x]]

cdar  [x] = cdr[car[x]]

cddr  [x] = cdr[cdr[x]]

list [x1;x2;...;xn] = cons [x1;cons[x2;...cons[xn;NIL]...]]
```

d). Additional auxiliary functions

```
reverse [x] = [null[x] → NIL;
               T → append[reverse[cdr[x]];
                          list[car[x]]]]

differ[x;y] = [null[x] → NIL;
               member[car[x] ; y] →
               differ[cdr[x]; y];
               T → cons[car[x]; differ[cdr[x];y]]]

union[x;y] = [null[x] → y;
              member[car[x];y] → union[cdr[x];y];
              T → cons[car[x]; union[cdr[x];y]]]

inter[x,y] = [null[x] → NIL;
              member[car[x];y] → cons[car[x];
                                      inter[cdr[x]; y]];
              T → inter[cdr[x]; y]]

incl[s;x] = equal[differ[s;inter[s;x]]; NIL]

eqset[x;y] = [incl[x;y] → incl[y;x];
              T → F]

count[x] = [null[x] → 0;
            T → add 1[count[cdr[x]]]]
```

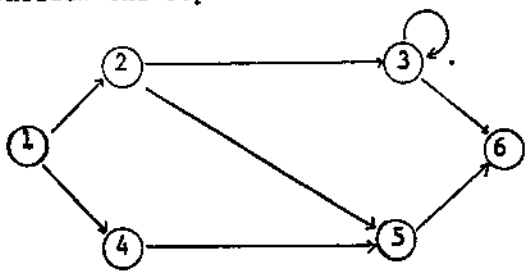
e). Functionals

```
maplist[x;f] = [null[x] → NIL;
                T → cons[f[x]; maplist[cdr[x];f]]]

map[x;f] = prog[[m];
     m: = x;
  LOOP[null[m] → return[NIL]];
     f[m];
     m: = cdr[m];
     go[LOOP]]

mapear[x;f] = [null[x] → NIL;
               T → cons[f[car[x]] ; mapcar[cdr[x];f]]]

mapc[x;f] = prog[[m];
      m: = x;
  LOOP[null[m] → return[NIL]];
     f[car[m]];
     m: = cdr[m];
     go[LOOP]]
```

2. GRAPH REPRESENTATION AND PROCESSING

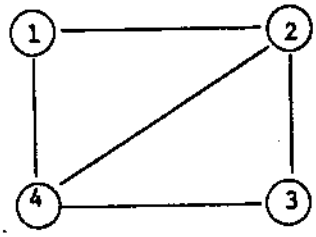
A graph G with n vertices is represented as a list with n sub-lists. The first element of each sub-list is the label of one of the n vertices; the remaining elements of a sub-list are the labels of the vertices adjacent to the given vertex.

Consider the representation of a directed graph.



G = ((1 2 4)(2 3 5)(3 3 6)(4 5)(5 6)(6))

An undirected graph may also be represented.



G = ((1 2 4)(2 1 4 3)(3 2 4)(4 1 2 3))

Although this representation is easy to specify and vizualize, some algorithms become simpler when applied to an expanded form. In the expanded form all lines are isolated as ordered pairs, the sinks being kept.

For the first example we have:

G' = ((1 2)(1 4)(2 3)(2 5)(3 3)(3 6)(4 5)(5 6)(6))

For the second example:

G' = ((1 2)(1 4)(2 1)(2 4)(2 3)(3 2)(3 4)(4 1)(4 2)(4 3))

A function to expand a graph, given in S-expression notation, is:

```
expand[g] = [null[g] → NIL;
             T → append[expand1[caar[g];cdar[g]];
                        expand[cdr[g]]]]

expand1[x;ℓ] = [null[ℓ] → cons[list[x] ; NIL];
                null[cdr[ℓ]] → list[cons[x;ℓ]];
                T → cons[list[x;car[ℓ]];
                         expand1[x;cdr[ℓ]]]]
```

After applying an algorithm having as input a graph in expanded form , it is convenient to reduce its output to the usual representation.

```
shrink[g] = [null[g] → NIL;
             null[cdr[g]] → g;
             T → shrink 1[car[g]; shrink[cdr[g]]]]

shrink 1[x;g] = [null[g] → list[x];
                 equal[car[x];caar[g]] →
                    shrink[cons[append[x;cdar[g]],cdr[g]];
                 T → cons[car[g],shrink 1[x,cdr[g]]]]
```

It is also convenient to represent a path (or chain) as a list, by simply giving the sequence of its vertices. In our example of a directed graph, one of the paths between vertices 1 and 6 is:

```
P = (1 2 5 6)
```

By a natural extension we may consider a set of paths under list form. In the same example, the set of elementary paths between 1 and 6 is:

```
LP = ((1 2 3 6)(1 2 5 6)(1 4 5 6))
```

To invert a path we may use:

```
invpath[p] = reverse[p]
```

and to invert all paths in a list of paths:

```
invlpath[ℓ] = [null[ℓ] → NIL;
               T → cons[reverse[car[ℓ]];
                        invlpath[cdr[ℓ]]]]
```

The set of vertices may also be presented as a list. In our example:

V = (1 2 3 4 5 6)

This set is obtained from the representation of the graph by:

```
vert[g] = [null[g] → NIL;
           T → cons[caar[g];vert[cdr[g]]]]
```

For an application to be described later, we shall need V in a slightly different form:

V' = ((1) (2) (3) (4) (5) (6))

which can be directly obtained from the graph representation by:

```
vert1[g] = [null[g] → NIL;
            T → cons[list[caar[g];vert1[cdr[g]]]]]
```

The application we have in mind is the inversion of a directed graph. It should be noted how the function performing this application utilizes previously defined functions.

```
invgraph[g] = shrink[append[vertl[g];invlpath
                                [expand[g]]]
```

In its turn, inverting a graph makes it easier to apply other algorithms. For instance, to obtain the set of vertices adjacent to a given vertex x , we use:

```
adjlist[x;g] = [null[g] → NIL;
                equal[x;caar[g]] → cdar[g];
                T → adjlist[x;cdr[g]]]
```

In case we are dealing with directed graphs, the result may be looked at as the list of successors of x. Now, the list of predecessors of x may be obtained by the same function, applied to the inverse graph:

```
predec[x;g] = adjlist[x;invgraph[g]]
```

Set theoretic operations are quite useful in graph processing. As a first example, we may obtain the set of vertices adjacent to a subset S of the vertices of a graph G:

```
adjset[s;g] = [null[s] → NIL;
               T → union[adjlist[car[s];g];
                         adjset[cdr[s];g]]]
```

Using another set theoretic operator we obtain the set of the lines of a graph:

line[g] = differ[expand[g];vertl[g]]

Now it is easy to count the elements of a graph:

nvert[g] = count[g]

nline[g] = count[line[g]]

Note that to count the vertices we took advantage of the representation that has been used for graphs.

Coming back to the application of set theoretic operators, we shall show how these operators are extended so as to have graphs as operands:

diffg[g1;g2] = diffg 1[differ[expand[g1] ;append
[expand[g2];vertl[g2]]]]

diffg 1[g] = shrink[union[vertl[invlpath[g]]; g]]

uniong[g1;g2] = shrink[union[expand[g1];expand[g2]]]

interg[g1;g2] = diffg[g1;diffg[g1;g2]]

Tests for checking for graph equality and whether a graph is contained in another, either as a sub-graph, partial graph, or partial sub -graph, are also needed.

```
eqg[g1;g2] = eqset[expand[g1];expand[g2]]

testsub[s;g] = [incl[vert[s];vert[g]] →
                    testsubl[s;g;vert[s]];
               T → F]

testsubl[s;g;v] = [null[s] → T;
        eqset[adjlist[caar[s];s],inter[adjlist[caar[s];
        g];v]] → testsubl[cdr[s];g;v];
        T → F]

testpar[p;g] = [eqset[vert[p];vert[s]] →
                incl[line[p];line[g]];
                T → F]

testpsub[ps;g] = eqg[interg[ps;g];ps]
```

As it should happen, the two graphs below will be taken as equal:

```
G1 = ((a b c) (b c) (c))
G2 = ((b c) (a c b) (c))
```

On the other hand, the tests for sub-graph, partial graph, and partial sub-graph do not exclude their being equal to the supposedly larger graph. For the same reason the test for a partial sub-graph does not exclude the case of a sub-graph or partial graph.

Two unary operations will be shown: undirected graph complementation (with respect to the complete graph) and rotation (cyclic permutation of the labels of the vertices).

```
complg[g] = complgl[g;vert[g]]

complgl[g;v] = [null[g] → NIL;
                T → cons[cons[caar[g];complg2
                              [car[g];v]];
                         complgl[cdr[g];v]]]

complg2[a;v] = [null[v] → NIL;
                member[car[v];a] → complg2[a;cdr[v]];
                T → cons[car[v];complg2[a;cdr[v]]]]

rot[g] = sublis[pairlis[vert[g];append[cdr[vert[g]];
                car[vert[g]]]; NIL];g]
```

Taking as an example the graph:

G = ((a b c) (b c d) (c d) (d))

then:

```
complg[G] = ((a d)(b a)(c a b)(d a b c))

rot[G] = ((b c d)(c d a)(d a)(a))
```

A graph may be modified by insertion or deletion of vertices or lines.

```
inserti[g;i] = append[g;list[list[i]]]

deli[g;i] = shrink[delil[expand[g];i]]

deli 1[e;i] = [null[e] → NIL;
               member[i;car[e]] → deli 1[cdr[e];i];
               T → cons[car[e];deli 1[cdr[e];i]]]

insertij[g;ij] = shrink[append[expand[g];list[ij]]]

delij[g;ij] = shrink[delij 1[expand[g];list[ij]]]

delij 1[e;ℓ] = [null[e] → NIL;
                member[car[e];ℓ] → delij 1[cdr[e];ℓ];
                T → cons[car[e];delij 1[cdr[e];ℓ]]]
```

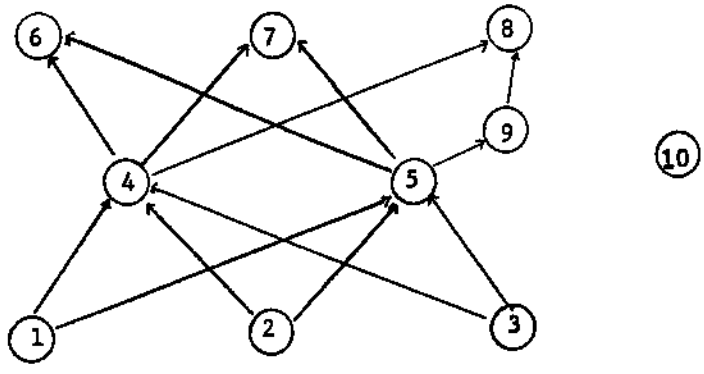
Another modification is changing the label of a vertex. It may also be necessary to collapse two vertices (x and y) into one and to assign a label (z) to it.

```
renam[y;x;g] = subst[y;x;g]

fuse[z;x;y;g] = sublis[cons[cons[x;z];
                        list[cons[y;z]]];
                        deli[g;cons[x;y]]]
```

Since a graph may have been modified, in certain moments it will be necessary to check for the presence of a given vertex or line in the graph.

```
testi[g;i] = member[i;vert[g]]

testij[g;i;j] = member[cons[i;j], line[g]]
```

Let us now consider a number of elements to be obtained from a graph as:



sinks = {6, 7, 8, 10}
sources = {1, 2, 3, 10}
isolated vertices = {10}
vertices above and including the cut defined by S = {4,5,6,7,8,9}
vertices below and including the cut = {4,5,1,2,3}
for S = {4,5}
a path from 5 to an element of the sink = (5,9,8)

These elements are respectively obtained by the functions:

```
sink[g] = [null[g] → NIL;
           null[cdar[g]] → cons[caar[g];sink[cdr[g]]];
           T → sink[cdr[g]]]

source[g] = sink[invgraph[g]]

isol[g] = inter[source[g];sink[g]]

pset[s;g] = pset 1[s;differ[s;sink[g]];g]

pset 1[s;d;g] = [null[d] → s;
                 T → union[s;pset[adjset[s;g];g]]]

mset[s;g] = pset[s;invgraph[g]]

sinkp [i;g] = [member[i;sink[g]] → list[i];
               T → cons[i;sinkp [car[adjlist[i;g];g]]]
```

So far we have restricted ourselves to the more elementary features of the LISP language. It is clear that others, as the PROG feature are also useful.

Very specially the use of functionals as map, mapc, maplist, and mapcar , making it possible to traverse lists,applying to each element a given process, corresponds to the effect of statements such as DO (FORTRAN , PL/I) or for (ALGOL).

To apply a certain function f to each vertex of a graph , we might use:

```
mapc[vert[g];f]
```

We shall present a realization of Warshall's algorithm, to determine all elementary paths between vertices x and y of a directed graph.

```
k[x;y;i] = [eq[i;0] → j[x;y];
            eq[x;z[i]] → k[z[i];y;sub 1[i]];
            eq[y;z[i]] → k[x;z[i];sub 1[i]];
            T → append[k[x;y;sub 1[i]];
                       concat[k[x;z[i];sub 1[i]],
                              k[z[i];y;sub 1[i]]]]]

j[a;b] = [member[cons[a;b];expand[g]] → cons[a;b];
          T → NIL]
```

```
z[ℓ] = z1[ℓ; 1; g]

z1[m;c;g] = [eq[m;c] → caar[g];
             T → z1[m;add 1[c];cdr[g]]]

concat[p;q] = [null[p] → NIL;
               null[q] → NIL;
               T → append[concat 1[car[p]; q],
                          concat[cdr[p];q]]]

concat 1[r;s] = [null[s] → nil;
                 T → cons[append[r;cdar[s]] ;concat 1[r;cdr[s]]]]
```

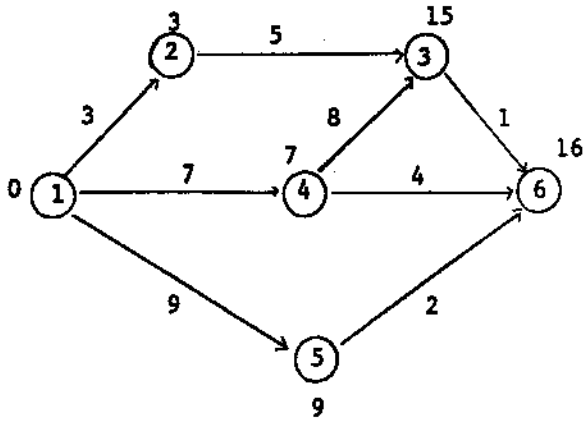
The function k should be called with:

```
k[x , y , nvert[g]]
```

where g is "global" to the program.

In order to treat valued graphs we consider two solutions.

The simplest one requires the definition of a list (V) structurally similar to G, in such a way that elements from G and V correspond to each other with respect to their position in these lists. G and V may be gathered in one list.



G = ((1 2 4 5)(2 3)(3 6)(4 3 6)(5 6)(6))

V = ((0 3 7 9)(3 5)(15 1)(7 8 4)(9 2)(1 6))

GV = (G V)

To search a value it is sufficient to traverse G and V in parallel.

```
val[x;gv] = val 1[x;car[gv];cadr[gv]]

val 1[x;g;v] = [null[g] → NIL;
                equal[x;car[g]] → car[v];
                T → val 1[x;cdr[g];cdr[v]]]
```

The other solution uses an attribute - value list (F) and is more flexible in the sense that it allows to associate more than one attribute - value pair to an element of the graph.

The list F is composed of ordered pairs, the second member of the pair being itself a list of ordered pairs: (graph element-

list of attribute - values belonging to element). Thus, if we call a an attribute of element x whose corresponding value is desired, we have:

val 2[a,x] = cadr[assoc[a;cadr[assoc[x;F]]]]

If x is the line (i j) then F is supposed to contain, among others, the pair:

F = (. . . ((i j) (. . . (a va) . . .)) . . .)

and the result of applying the function will be va.

It should be remarked that val 2 does not include g is its argument list. This is related to the fact that F can be shared by several graphs.

3. REMARKS

The student is urged to use these materials as a starting point. He could, for example:

a. extend the package by adding other functions;

b. recode certain functions in a more efficient way;

c. describe and test non-trivial algorithms, using the original or an extended package.

Since LISP - like list - processing has been enbedded in several high level languages (CPL , FORMULA ALGOL, several FORTRAN , ALGOL, and PL/I implementations, etc.), this package could equally well be added to them.

BIBLIOGRAPHY

- Mc Carthy, J - "LISP 1.5 Programmer's Manual" - the MIT Press - 1966.

- Crespi - Reghizzi, S. and Morpurgo, R. - "A Language for Treating Graphs" - CACM 13/5/1970.

- Derniame, J and Pair, C - "Problèmes de cheminement dans -les Graphes" - Dunod - 1971.

- Berge, C - "Théorie des Graphes et ses Applications" - Dunod - 1950.

A P P E N D I X

LISTING OF LISP PROGRAMS AS RUN ON THE

IBM 1620 COMPUTER OF THE CENTRO BRASILEIRO DE PESQUISAS FÍSICAS

```
(DEFINE

     (EQUAL(LAMBDA(X Y)(COND
           ((ATOM X)(EQ X Y))((ATOM Y)(OR))
           ((NULL X)(NULL Y))((NULL Y)(OR))
           ((EQUAL(CAR X)(CAR Y)) (EQUAL (CDR X)(CDR Y)))
           ((AND)(OR)) )))

     (SUBST(LAMBDA(X Y Z)(COND
           ((NULL Z)(LIST))
           ((EQUAL Y Z) X)
           ((ATOM Z) Z)
           ((AND)(CONS (SUBST X Y (CAR Z))
                       (SUBST X Y (CDR Z)) )) )))

     (APPEND(LAMBDA(X Y)(IF  (NULL X)
                             Y
                             (CONS (CAR X) (APPEND (CDR X) Y))  )))

     (MEMBER(LAMBDA(X Y)(COND
           ((NULL Y) (OR))
           ((EQUAL X (CAR Y)) (AND))
           ((AND)(MEMBER X (CDR Y)) ) )))

     (PAIRLIS(LAMBDA(X Y A)(IF (NULL Y)
                               A
                               (CONS (CONS (CAR X)(LIST(CAR Y)))
                                     (PAIRLIS (CDR X)(CDR Y) A) ) )))

     (ASSOC(LAMBDA(X A)(IF  (EQUAL (CAAR A) X)
                            (CAR A)
                            (ASSOC X (CDR A)) )))

     (SUBLIS(LAMBDA(A Y)(COND
           ((NULL Y)(LIST))
           ((ATOM Y)(SUB2 A Y))
           ((AND)(CONS (SUBLIS A (CAR Y))
                       (SUBLIS A (CDR Y)) ) )) )

     (SUB2(LAMBDA(A Z)(COND
           ((NULL A) Z)
           ((EQ (CAAR A) Z) (CADAR A))
           ((AND)(SUB2 (CDR A) Z)) )))

     (ADD1(LAMBDA(X)(ADD X 1)))

     (SUB1(LAMBDA(X)(SUB X 1)))

     (MAPLIST(LAMBDA(X F)(IF (NULL X)
                             (LIST)
                             (CONS (F X) (MAPLIST(CDR X) F)) )))

     (MAP(LAMBDA(X F)(IF  (NULL X)
                          (LIST)
                          (AND (F X) (MAP(CDR X) F)) )))
```