ON IMPLEMENTING GRAPH REPRESENTATIONS

Csaba J. Egyhazy
Computer Science Department
Virginia Polytechnic Institute & State University
Falls Church, VA  22042

CS830006
ON IMPLEMENTING GRAPH REPRESENTATIONS

Csaba J. Egyhazy
Computer Science Department
Virginia Polytechnic Institute & State University
2990 Telestar Ct.
Falls Church, Va. 22042

Summary. The three most common graph representations, namely the adjacency matrix, one way adjacency lists and adjacency multilist, are implemented in PASCAL and their performance evaluated for twelve graphs typical to computer network configurations.

We show that both adjacency multilist and one way adjacency lists are preferred over the adjacency matrix representation. Although their implementation is slightly more complicated it outperforms the latter by a factor of at least 5.

1. Introduction

The choice of a particular graph representation is an important design decision in implementing algorithms to perform functions on graphs. We know intuitively that the performance characteristics of software, given the same set of simple, connected and undirected graphs will be different for each representation implemented. It is this claim that we are set forth to prove. In the next section we will define the terminology and notation
for the three most common graph representations: Adjacency matrix, adjacency list and adjacency multilist, as well as indicating two possible implementations for each. Section 3 is devoted to the implementation strategies for each graph representation using the programming language PASCAL. Sample programs to initialize the structure for each graph representation are given.

In Section 4 we describe the experiment conducted to measure the time performance of each representation implemented for a single set of 12 graphs. The results of the test runs are reported and discussed. The last Section is devoted to comments and concluding remarks.

2. Graph Representations

Since many problems or instances of problems can best be visualized and analyzed by simple, connected and undirected graphs, efficient methods for representing them are desirable, both in terms of processing time and storage [1] [2].

In an adjacency matrix representation of a graph G the matrix entry \((v_i, v_j)\) equals one if and only if there is an edge between vertex \(v_i\) and vertex \(v_j\). Since for this investigation only simple graphs are considered, the main diagonals in the adjacency matrix will always equal zero. Accordingly, in the adjacency matrix representation of a graph, only the upper triangle of the adjacency matrix will be used to represent the graph.
The same adjacency matrix could, if PL/I or FORTRAN are used, be implemented in two different ways. In the first implementation the matrix entries would consist of single bits while in a second implementation the matrix entries would consist of one word of storage (w bits). Computer storage will be conserved in the first implementation but more processing time will be required to access the matrix entries. In the second implementation the processing time to access the entries of the adjacency matrix will be less but more storage will be required to represent the graph.

In the adjacency list representation of a graph G there will be one list for each vertex \( v_i \) in G. In each list the nodes represent the specific vertices adjacent to vertex \( v_i \). Each node consists of two fields, INFO and LINK. The INFO field contains the identifier of the next vertex that is also adjacent to vertex \( v_i \).

Again, if PL/I or FORTRAN were to be used, the same adjacency list would be implemented in two different ways. In the first implementation the adjacency list will consist of \( n \) header nodes each of \( \log n \) bits and will consist of \( 2e \) list nodes each of \( \log n = \log e \) bits. In the second implementation each of the \( n \) header nodes would consist of \( w/2 \) bits (a halfword) and each of the \( n \) list nodes will consist of \( w \) bits (\( w/2 \) bits for the INFO field and \( w/2 \) bits for the LINK field). These two implementations correspond to the tradeoffs noted previously concerning storage utilization and processing time.
The adjacency lists method for representing graphs is compared by Itali [3] to the information theoretic lower bounds, and it is shown to be optimal in many instances. They also propose a procedure that works in linear time and produces a representation more compact than the adjacency list, which requires at most $3/2 n \log n + O(n)$ bits.

In the adjacency multilist representation of a graph $G$ there will be one list for each vertex in $G$. Each list has one node for each edge in $G$. The node represents the edge that has vertices $v_i$ and $v_j$ incident to a specific edge. Thus each node will be a member of two lists; one a member for vertex $v_i$ and one a member for vertex $v_j$.

Each list node has five fields. Let $M$, INFOVI, INFOVJ, LINKVI, and LNIKVJ denote these five fields. The first field $M$ is an indicator of whether the edge has been inspected. The second field INFOVI contains the identifier of vertex $v_i$ that is incident to the edge represented by the node. The third field INFOVJ contains the identifier of vertex $v_j$ for which both $v_i$ and $v_j$ are incident to the edge represented by the node. The fourth field LINKVI contains a pointer to another node for which an edge is incident to vertex $v_i$. The fifth field LNIKVJ contains a pointer to another node for which an edge is incident to vertex $v_j$. Thus the node structure is as follows:

```
+--------+--------+--------+--------+--------+
|   M    | INFOVI | INFOVJ | LINKVI | LNIKVJ |
+--------+--------+--------+--------+--------+
```
The storage requirements are the same as for adjacency lists except for the addition of the mark bit M. The same adjacency multilist could, if PL/I or Fortran is used, be implemented in two different ways. In the first implementation the adjacency multilist would consist of n header nodes each of \( \log n \) bits and each list will consist of \( e \) nodes (one node for each edge). The storage for the five fields of each list node is as follows: one bit for the inspection off field M, \( \log n \) bits each for fields INFOV1 and INFOV2, and \( \log e \) bits each for fields LINKV1 and LINKV2. In the second implementation the same adjacency multilist would consist of \( n \) header nodes each of \( w/2 \) bits (a half-word) and each list will consist of \( e \) nodes. The storage for the five fields of each list node is as follows: one bit for the inspection field M, \( w/2 \) bits each for fields INFOV1 and INFOV2, and \( (w-1)/2 \) bits for fields LINKV1 and LINKV2. Thus in the second implementation each list node requires a double word of storage.
3. Implementation of Graph Representations

All three graph representations described above were implemented in both IBM and DEC computers using the programming language PASCAL. The choice of PASCAL eliminated any consideration to alternative implementations within a particular graph representation. Therefore, the emphasis was in finding the proportion of CPU time consumed in representing graph G as opposed to storage requirements.

Each of the three graph representations requires a different design approach when implemented. The adjacency matrix representation is certainly the easiest to implement, since all it requires is an input via a disk file and a structure initialization of the form:

```
PROGRAM; ADJACENCY MATRIX REPRESENTATION

CONST
MAX_N = 45;  {NUMBER OF VERTICES}

TYPE

VERTEX = 0 .. MAX_N;
ADJ MATRIX = ARRAY [1 .. MAX_N, 1 .. MAX_N] OF VERTEX;

VAR

N, V_E, V_W, MATRIX_A; ADJ_MATRIX, TEMP; VERTEX, SUM_ARROW_ROW;
VERTEX_ARRAY

BEGIN
READLN(N);  [INITIALIZE ADJACENCY MATRIX]
FOR TEMP = 1 to N DO
  SUM_ARROW_ROW[TEMP] := 0
FOR V_E = 1 to N DO
  BEGIN
    READ (MATRIX_A[V_E, V_W]);
    SUM_ARROW_ROW[V_E] := SUM_ARROW_ROW[V_E] + MATRIX_A[V_E, V_W];
END;
```

PROGRAM 1
The adjacency list implementation is derived from the adjacency matrix representation, since the n rows of the adjacency matrix are represented as n linked lists. The nodes in a particular list represent the vertices that are adjacent to that list. The record structure and its initialization is as follows:

```pascal
PROGRAM ADJACENCY_LIST_REPRESENTATION
CONST
MAX_N = 45;  // NUMBER OF VERTICES

TYPE
  NODE_POINTER = ^VERTEX_NODE;

  VERTEX_NODE = RECORD
    V_INFO : VERTEX_RANGE;  // IN THE ADJACENCY LISTS
    V_LINK : NODE_POINTER  // STRUCTURE OF EACH NODE
  END;

VAR
  V_INDEX, NUM_NODES, SUM_ARR_ROW, INDEX_VAR, ADJ_PTR, N
  TOP_ADJ : FIRST_POINTER;  // AN ARRAY OF POINTERS THAT POINT TO
                              // THE TOP OF EACH ADJACENCY LIST
  BOT_ADJ_PTR: NODE_POINTER;

BEGIN
  | INITIALIZE YOUR STRUCTURES |
  READLN (N);
  FOR V_INDEX := 1 TO N DO BEGIN
    READ (NUM_NODES);  // READ THE NUMBER OF ADJACENT VERTICES
    SUM_ARR_ROW (V_INDEX) := NUM_NODES;
    INDEX_VAR := NUM_NODES;
    TOP_ADJ (V_INDEX) := NIL;
    WHILE (NUM_NODES > 0) DO BEGIN
      NEW (ADJ_PTR);
      READ (ADJ_PTR^, V_INFO);
      IF (INDEX_VAR = NUM_NODES) THEN BEGIN
        TOP_ADJ (V_INDEX) := ADJ_PTR;
        BOT_ADJ_PTR := ADJ_PTR;
        END ELSE BEGIN
          BOT_ADJ_PTR^.V_LINK := ADJ_PTR;
          BOT_ADJ_PTR := ADJ_PTR;
          END;
        ADJ_PTR^.V_LINK := NIL;
        NUM_NODES := NUM_NODES - 1
      END;
    READLN
  END;
END;
```

PROGRAM 2
The first problem encountered in implementing the adjacency multilist is determining in what INFOV field (i or j) the vertex V_Z being examined is located. The use of an inspection indicator of the form I_INDEX and J_INDEX permits us to check for the validity of the statement LOC_V_Z = I_INDEX. If the equality holds V_Z is in the INFOVI field, otherwise V_Z is in the INFOVJ field. Furthermore, a flag of type boolean is used to indicate if the vertex currently being focused on is in the INFOVI component of the node or the INFOVJ component of the node.

The adjacency multilist representation is actually established by means of an array of top pointers and two temporary pointers, PTRT1 and PTRT2. For the first record, the M field is set to zero while the two INFO fields are read in. Finally, both LINK fields are given the value nil. As we set up our second record the LINKVI field of the predecessor will be pointing to this second record if the vertex is common to both edges. The LINKVJ field on the other hand will be nil until a record is established that contains that vertex. One interesting aspect of this implementation is that the vertex can appear in either the I or the J INFO field. A flag of type boolean is used throughout to test for the exact location of the vertex. The complete code for the establishment of an adjacency multilist representation is given by Program 3.
PROGRAM: ADJACENCY MULTILIST REPRESENTATION
CONST
MAX_N = 45    ( NUMBER OF VERTICES )

TYPE
FLAG_FIELD = 0, 1;
VERTEX_RANGE = 0, ..., MAX_N;
STATE_NODE = RECORD
V_INFO: VERTEX_RANGE
V_INFO: VI_TABLE_INDEX, POINTER
END;

NODE_POINTER = * VERTEX_NODE

THIS IS THE STRUCTURE OF THE ADJACENCY MULTILIST

VERTEX_NODE = RECORD
INSPECT : FLAG_FIELD;  \ THIS FIELD INDICATES
                    \ WHETHER THE NODE HAS
V_INFO_I : VERTEX_RANGE; \ BEEN INSPECTED
V_INFO_J : VERTEX_RANGE; \ THIS FIELD HOLDS ONE
V_INFO_J : VERTEX_RANGE; \ VERTEX IDENTIFIER
V_LINK_I : NODE_POINTER; \ THIS FIELD POINTS TO
V_LINK_J : NODE_POINTER \ THE NEXT NODE INCIDENT
                    \ TO VERTEX V_INFO_I
V_LINK_J : NODE_POINTER \ THIS FIELD POINTS TO
                    \ THE NEXT NODE INCIDENT
END;

VAR
V, W, N, NUM_NODES, ML_PTR, INDEX_VAR: VERTEX_RANGE
HM_ARR: ARRAY VERTEX_RANGE OF HM_PTR;
TOP_ML, FIRST_POINTER: \ CONTAINS POINTERS TO THE
                    \ TOP EDGE FOR EACH VERTEX
BOT_ML_PTR: NODE_POINTER

\ ESTABLISH THE ADJACENCY MULTILISTS
FOR I, N := 1 TO N DO
BEGIN
    READ (NUM_NODES);
    INDEX_VAR := 0; NUM_NODES;
    WHILE (NUM_NODES > 0) DO
    BEGIN
        RECHG
        NEW (ML_PTR);
        ML_PTR.INSPECT := 0;
        ML_PTR.V_INFO_I := V_W;
        HM_ARR (ML_PTR.V_INFO_I) := 1;
        IF (SOURCE_VAR = NUM_NODES) THEN
            IF (TOP_POINTER = NIL) THEN
                TOP_ML := ML_PTR
            ELSE
                IF (TOP_POINTER IS NOT ASSIGNED) THEN
                    TOP_POINTER := ML_PTR
                ELSE
                    IF (TOP_POINTER IS ASSIGNED) THEN
                        SEARCH THE LIST OF V.W FOR
                        LINK FIELD J WHICH HAS NIL VALUE
                        PERFORM THE SEARCH FROM THE TOP
                        POINTER ON DOWN THE LIST
                        TEST IF IT IS THE FIRST NODE
                        THAT HAS A LINK FIELD J WITH
                        A NIL VALUE
                        IF (TOP_ML.V_W = "V.W") THEN
                            TOP_ML.V_W := ML_PTR
                        ELSE
                            SEARCH THE LIST FOR A NODE WITH
                            A LINK J FIELD VALUE OF NIL
                            INITIALSE A SEARCH PTR
                            TEMP_PTR := ML_PTR
                            WHILE TEMP_PTR.V_LINK_J =NIL
                            DO TEMP_PTR := TEMP_PTR.V_LINK_J
                            SET V.W.FIELD WHICH HAS
                            NIL TO ML_PTR POINTER
                            TEMP_PTR.V_W.V_LINK_J := ML_PTR
                        END;
                        BOT_ML_PTR := ML_PTR
                        ELSE
                            INDEX_VAR := NUM_NODES
                        END
                    END;
                END;
            END;
        END;
    END;
END;

PROGRAM 3

-9-
4. **Experiment and Test Results**

A preslected sample set of graphs from typical computer network configurations [4,5] were produced for the test runs. These graphs are illustrative examples of the following small network structures:

A) **Distributed Network**: This graph has 11 vertices and is configured so that no vertex is further away than three edges from any other vertex (Figure A).

B) **Point to Point Long Haul Network**: This graph has 41 vertices, with 8 vertices forming a ring structure. One of the ring vertices is connected to a subnet structure while the others are the roots of tree structures (Figure B).

C) **Double Ring Network**: This graph has a double ring like structure consisting of 12 vertices. Each ring vertex is connected to one and only one vertex of the other ring (Figure C).

D) **Intersecting Loop**: This graph has two distinct cycles with one vertex in common (Figure D).

E) **Two Level Hierarchy with Five Regions**: This graph, comprised of 17 vertices has two levels of hierarchy and five distinct regional topology (Figure E).
F) Interconnection Network of Networks: This graph has a complete subgraph of five vertices as the interconnecting network and five separate network structures connected to the complete subgraph by only one edge (Figure F).

G) Regular Network Representation: This graph has 20 vertices with every vertex of degree four (Figure G).

H) Irregular Network Representation: This graph has 12 vertices with one star tree configuration as one subgraph (Figure H).

I) Linked Network Representation: This graph has 24 vertices with a "unibus" structure to which five subgraphs are connected (Figure I).

J) Distributed Network Representation: This graph has 128 vertices with a complete subgraph and four star like configurations. This network is similar in structure to the DATAPAC Packet Switched Network [7]. The graph is given in Figure K.

K) Subnet 1: This planer graph has 8 vertices forming to squares with one face in common.

L) Subnet 2: This graph is comprised of 15 vertices.

All of the above graphs are given pictorially in Appendix A.
For each of the above graphs we measured the CPU minutes consumed in the execution of a particular PASCAL program, hereby referred to as the "initialization" program, for each of the three graph representations.

The virtual CPU minutes consumed was used to measure CPU minute consumption on the IBM 4341 VM/CMS. This quantity was obtained by involving a QUERY TIME command just prior to execution, immediately after execution and then subtracting the first from the second. On the other hand, the total CPU minutes consumed was used for measuring CPU minute consumption in the VAX-11/780 VMS. This quantity was obtained by invoking a SHOW STATUS command just prior to executing a program, immediately after execution and then subtracting the first from the second. Since the total CPU time on both machines were influenced by the load on the system at the time of execution, the measures of CPU minute consumption were averaged over three executions, performed at different times of the day, of the same program.

Tables I, II and III give for each graph the IBM 4341 VM/CMS virtual CPU minute and VAX-11/780 VMS CPU minutes consumption values using an adjacency matrix, adjacency list and adjacency multilist graph representations are respectively.
TABLE I

a. IBM 4341 VM/CMS VIRTUAL CPU MINUTES CONSUMED IN IMPLEMENTING THE ADJACENCY MATRIX GRAPH REPRESENTATION

<table>
<thead>
<tr>
<th>GRAPHS</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZATION</td>
<td>.05</td>
<td>.23</td>
<td>.05</td>
<td>.07</td>
<td>.07</td>
<td>.20</td>
<td>.07</td>
<td>.05</td>
<td>.10</td>
<td>.08</td>
<td>.07</td>
<td>.04</td>
</tr>
<tr>
<td></td>
<td>.05</td>
<td>.23</td>
<td>.06</td>
<td>.07</td>
<td>.07</td>
<td>.20</td>
<td>.07</td>
<td>.05</td>
<td>.11</td>
<td>.07</td>
<td>.06</td>
<td>.04</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>.05</td>
<td>.2333</td>
<td>.0533</td>
<td>.0667</td>
<td>.07</td>
<td>.2033</td>
<td>.07</td>
<td>.0577</td>
<td>.1067</td>
<td>.0733</td>
<td>.0667</td>
<td>.0433</td>
</tr>
</tbody>
</table>

b. VAX-11/780 CPU MINUTES CONSUMED IN IMPLEMENTING THE ADJACENCY MATRIX GRAPH REPRESENTATION

<table>
<thead>
<tr>
<th>GRAPHS</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZATION</td>
<td>.29</td>
<td>.66</td>
<td>.30</td>
<td>.32</td>
<td>.34</td>
<td>.39</td>
<td>.33</td>
<td>.30</td>
<td>.40</td>
<td>.34</td>
<td>.39</td>
<td>.29</td>
</tr>
<tr>
<td></td>
<td>.31</td>
<td>.66</td>
<td>.30</td>
<td>.31</td>
<td>.33</td>
<td>.62</td>
<td>.31</td>
<td>.29</td>
<td>.41</td>
<td>.37</td>
<td>.32</td>
<td>.25</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>.29</td>
<td>.6633</td>
<td>.2967</td>
<td>.32</td>
<td>.0337</td>
<td>.6167</td>
<td>.3167</td>
<td>.30</td>
<td>.4067</td>
<td>.3467</td>
<td>.31</td>
<td>.2767</td>
</tr>
</tbody>
</table>

TABLE II

IBM 4341 VM/CMS VIRTUAL CPU MINUTES CONSUMED IN IMPLEMENTING THE ADJACENCY LIST GRAPH REPRESENTATION

<table>
<thead>
<tr>
<th>GRAPHS</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZATION</td>
<td>.05</td>
<td>.11</td>
<td>.05</td>
<td>.05</td>
<td>.07</td>
<td>.11</td>
<td>.06</td>
<td>.05</td>
<td>.07</td>
<td>.06</td>
<td>.06</td>
<td>.05</td>
</tr>
<tr>
<td></td>
<td>.05</td>
<td>.11</td>
<td>.05</td>
<td>.05</td>
<td>.06</td>
<td>.11</td>
<td>.07</td>
<td>.05</td>
<td>.08</td>
<td>.06</td>
<td>.06</td>
<td>.05</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>.05</td>
<td>.11</td>
<td>.05</td>
<td>.05</td>
<td>.0633</td>
<td>.1067</td>
<td>.0667</td>
<td>.05</td>
<td>.0733</td>
<td>.06</td>
<td>.06</td>
<td>.05</td>
</tr>
</tbody>
</table>

VAX-11/780 CPU MINUTES CONSUMED IN IMPLEMENTING THE ADJACENCY LIST GRAPH REPRESENTATION

<table>
<thead>
<tr>
<th>GRAPHS</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZATION</td>
<td>.27</td>
<td>.33</td>
<td>.25</td>
<td>.29</td>
<td>.27</td>
<td>.31</td>
<td>.29</td>
<td>.26</td>
<td>.30</td>
<td>.29</td>
<td>.28</td>
<td>.27</td>
</tr>
<tr>
<td></td>
<td>.27</td>
<td>.31</td>
<td>.29</td>
<td>.27</td>
<td>.30</td>
<td>.33</td>
<td>.27</td>
<td>.26</td>
<td>.29</td>
<td>.28</td>
<td>.28</td>
<td>.27</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>.27</td>
<td>.3327</td>
<td>.2767</td>
<td>.28</td>
<td>.29</td>
<td>.32</td>
<td>.29</td>
<td>.2567</td>
<td>.2967</td>
<td>.2767</td>
<td>.28</td>
<td>.2733</td>
</tr>
</tbody>
</table>
### TABLE III

<table>
<thead>
<tr>
<th>GRAPHS</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZATION</td>
<td>.05</td>
<td>.08</td>
<td>.05</td>
<td>.05</td>
<td>.09</td>
<td>.06</td>
<td>.05</td>
<td>.07</td>
<td>.05</td>
<td>.06</td>
<td>.04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.05</td>
<td>.08</td>
<td>.05</td>
<td>.05</td>
<td>.06</td>
<td>.10</td>
<td>.06</td>
<td>.04</td>
<td>.06</td>
<td>.05</td>
<td>.04</td>
<td></td>
</tr>
<tr>
<td>AVERAGE</td>
<td>.0467</td>
<td>.0833</td>
<td>.05</td>
<td>.05</td>
<td>.0933</td>
<td>.0933</td>
<td>.0567</td>
<td>.0467</td>
<td>.0667</td>
<td>.0533</td>
<td>.0533</td>
<td>.04</td>
</tr>
</tbody>
</table>

### TABLE IV

<table>
<thead>
<tr>
<th>Summary Across Twelve Graphs of the CPU Minutes Consumed in Implementing All Three Graph Representations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM 4341 VM/CMS</strong></td>
</tr>
<tr>
<td>Adjacency Matrix</td>
</tr>
<tr>
<td>Adjacency List</td>
</tr>
<tr>
<td>Adjacency Multilist</td>
</tr>
<tr>
<td><strong>VAX - 11-780 VMS</strong></td>
</tr>
<tr>
<td>Adjacency Matrix</td>
</tr>
<tr>
<td>Adjacency List</td>
</tr>
<tr>
<td>Adjacency Multilist</td>
</tr>
</tbody>
</table>
Three different runs per representation were made and an average computed. Table IV summarizes the results of Tables I, II, and III. It shows that the adjacency multilist graph representation yielded the minimal expenditure of virtual CPU minutes, the one way adjacency list yielded the next lowest while the adjacency matrix representation was by far the one with the highest expenditure of virtual CPU minutes. The difference in CPU minute consumption between the adjacency multilist and the one way adjacency list representation is minimal compared to the CPU minutes consumed in implementing the adjacency matrix representation. The same ranking of the three representations was obtained for both IBM 4341 and VAX-11/780 machines. As expected, the implementation of all three representations on the VAX-11/780 consumed, on the average, five times more CPU time than the equivalent run on the IBM 4341. Which shows that the smaller the computer we have available the more critical, in terms of both time and store, the choice of a graph representation becomes.

One of the main findings of this research effort was that the choice of graph representation substantially influences the CPU time consumed in initializing the structure in question. The impact of the representation is also felt in time it takes to traverse a graph as found in [6]. There we demonstrated that across algorithms for generating the set of fundamental cycles the adjacency multilist and the one way adjacency list graph representation implementations were found to require considerable less CPU time than did the adjacency matrix graph representation.
Furthermore, we found that, for the same set of graphs than those
given in this paper, 40 to 60 percent of the total time was con-
sumed in graph representation and traversal of the structure,
while the rest was consumed in executing the algorithm. There-
fore, the argument that choosing a representations other than the
adjacency matrix for graphs with more than 20 vertices is so far
convincing. Ongoing research, at Virginia Polytechnic Institute
and State University, hopes to further ascertain the impact of
alternative graph representations in algorithm design and program
performance.

References

1. Horowitz, E., Shani, S. "Fundamentals of Data Structures"

2. Aho, A., Hopcroft, J., Ullman, J. "Data Structures and Algo-
rithms" Addison-Wesley, 1983.


to Generate Fundamental Cycles for Twelve Different Graphs"
Technical Report #CS82012, Department of Computer Science,
Virginia Polytechnic Institute and State University, December
1982.
APPENDIX A

A SPECIAL CLASS OF GRAPHS

This appendix contains the topologies for the twelve graphs used in this research effort.
FIGURE A
Distributed Network

FIGURE B
Point to Point Long Haul Network
FIGURE C
Double Ring Like Network

FIGURE D
Intersecting Loop

FIGURE E
Subnet 2
FIGURE F
Two Level Hierarchy With Five Regions

FIGURE G
Interconnection Network of Networks
FIGURE H
Regular Network Representation

FIGURE I
Irregular Network Representation
FIGURE J
Linked Network Representation

FIGURE K
DATAPAC Packet Switched Network

FIGURE L
Subnet 1