AN EDGE-ORIENTED ADJACENCY LIST FOR UNDIRECTED GRAPHS

by

L. K. Chen, B. S. Ting and A. Sangiovanni-Vincentelli

Memorandum No. ERL-M589

18 May 1976

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

AN EDGE-ORIENTED ADJACENCY LIST FOR UNDIRECTED GRAPHS*

L. K. Chen, B. S. Ting and A. Sangiovanni-Vincentelli†

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

ABSTRACT

A new data structure, called the Edge-Oriented Adjacency List (EOAL), for representing undirected graphs is presented. It provides more information on the edges and requires less storage space than the conventional adjacency list. Furthermore, it is superior than the conventional adjacency list in both insertion and deletion operations.

† On leave from Instituto di Elettrotecnica ed Elettronica, Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milano, Italy.

* Research sponsored by the National Science Foundation Grant ENG74-06651-A01 and the Joint Services Electronics Program Contract F44620-71-C-0087

1. Introduction

One of the most commonly used data structures for representing graphs is a list structure--the adjacency list [1]. It is particularly suited for graph manipulations where searchings and sortings are involved [2]. It has been incorporated in depth-first-search algorithms [3,4], breadth-first-search algorithms [5] and maximum flow problems [6]. It is also implemented in some graph programming language like GEA [7]. However, there are two major disadvantages--the duplication of data and the lack of edge identificiation property--associated with this conventional adjacency list when used in conjunction with undirected graphs. We shall propose a modified version of this conventional adjacency list, called the Edge-Oriented Adjacency List and denoted by EOAL, which not only takes care of the above disadvantages but also proves to be superior in graph modification where insertions and deletions are needed.

In Sec. 2, we shall review the conventional adjacency list briefly and discuss its disadvantages. In Sec. 3, we shall introduce the EOAL structure and discuss its advantages. Throughout this paper, we shall use the same example to illustrate the differences between the conventional adjacency list and EOAL.

2. Conventional Adjacency List

The conventional adjacency list is defined as follows [1]:

Let $G = (V,E)$ denote an undirected graph; for each vertex $u \in V$, we provide a list containing all the vertices $v \in V$ such that $\{u,v\}$[1] $\in E$ (i.e., there is an edge connecting vertices u and v). Observe that each

[1] $\{u,v\}$ denotes an unordered pair of vertices u and v whereas (u,v) denotes an ordered pair of vertices u and v.

edge $\{u,v\}$ is represented twice in the lists, i.e., in the lists associated with vertices u and v.

Let us assume that all vertices are numbered from 1 to $|V|$[2], i.e., $V = \{v_1, v_2, \ldots, v_{|V|}\}$. In the most common programming languages (e.g., FORTRAN, ALGOL), the implementation of the conventional adjacency list requires $|V| + 4|E|$ cells.[3] For example, the adjacency list for the graph shown in Fig. 1 is represented in Fig. 2 where row 1 of Fig. 2 states that the vertex v_1 is adjacent to vertices v_2 and v_4. A total of $|V| + 4|E| = 4 + 4 \times 5 = 24$ cells are needed.

One disadvantage of this adjacency list is that if there are k data (e.g., name, cost, capacity, etc.) associated with each edge of G, an additional $2k|E|$ cells are needed as shown in Fig. 3. Observe that we have represented each of the $k|E|$ data twice.

Another disadvantage of this adjacency list is its lack of the "edge identification" property. We shall discuss briefly this property now. During the execution of an edge (u,v) in an undirected graph, we want to devise some scheme that enables us to avoid any future execution of the same edge (v,u). This is the so-called edge identification problem.

There are in general three approaches for solving this problem. The first approach is to modify the adjacency lists associated with vertices u and v right after the execution of edge (u,v). This requires sequential search through the adjacency list of v. The second approach is to devise some simple test criterion (as in [3]) such that, once (u,v) is executed, (v,u) will fail the test hence avoiding the execution

[2] $|S|$ denotes the cardinality of the set S.

[3] By a cell we mean a storage space containing either a number or a name associated with a vertex or an edge of G.

of (v,u) at a later stage. It has to be noted, however, that very often such a test criterion is hard to find or it may be computationally involved. The last approach is to add $3|E|$ additional cells as shown in Fig. 4. The LABEL array in Fig. 4 is then the simple test criterion described earlier. Before the execution of a particular edge, the corresponding space in LABEL is checked to see whether it has been "marked." After the execution of an "unmarked" edge, the corresponding space in LABEL is then "marked."

If we need both the k data associated with each edge and the edge identification property, a total of $|V| + 4|E| + 2k|E| + 3|E| = |V| + (2k+7)|E|$ cells are needed. The number of cells can be partially reduced by using the edge identification scheme along with a table of size $(k+1) \times |E|$ as shown in Fig. 5 where a total of $|V| + 4|E| + 2|E| + (k+1)|E| = |V| + (k+7)|E|$ cells are needed.

3. <u>Edge-Oriented Adjacency List</u>

Now, we shall propose a modified version of the conventional adjacency list, called the Edge-Oriented Adjacency List (i.e., EOAL), to further reduce the number of needed cells. We shall first discuss EOAL without any data associated with edges. Then, edge data will be added to EOAL structure.

Let EOAL denote a 1-dimensional array of $|V| + 4|E|$ cells (see Fig. 6(a)). Let the first $|V|$ cells be denoted by VA (i.e., Vertex Array) and let the remaining $4|E|$ cells be denoted by EA (i.e., Edge Array). The ith cell of VA, denoted by VA_i, contains the pointer associated with vertex v_i.

Each consecutive four cells in EA corresponds to an edge e_j and is denoted by EA_j, $j = 1,2,\ldots,|E|$. The first cell in EA_j is denoted by EF_j, containing one of the two end vertices associated with edge e_j. The second cell in EA_j is denoted by EFP_j, containing the pointer for the vertex in EF_j. The third cell in EA_j is denoted by ET_j, containing the remaining end vertex associated with e_j. The fourth and last cell in EA_j is denoted by ETP_j, containing the pointer for the vertex in ET_j.

The entries of VA_i, EF_j, EFP_j, ET_j and ETP_j can be found through the following construction algorithm. Let $V = \{v_1, v_2, \ldots, v_{|V|}\}$, $E = \{e_1, e_2, \ldots, e_{|E|}\}$ and let $\{v_{f_j}, v_{t_j}\}$ denote the two end vertices associated with edge e_j. Let CA denote a 1-dimensional storage array of size $|V|$ where each of its components, denoted by CA(i), contains the Current Address of the end-of-list sign for the adjacency list of the corresponding vertex v_i. We can then summarize the construction of EOAL by the flowchart shown in Fig. 7. For our example in Fig. 1, the step by step construction of the EOAL appears in Figs. 8(a)-(f). Observe that in EOAL pointers are stored in VA_i, EFP_j and ETP_j. We can extract the list structure stored in Fig. 8(f) as shown in Fig. 8(g) which is identical to the conventional adjacency list shown in Fig. 2.

Let us recall that in Fig. 4 the edge identification implementation requires $3|E|$ additional cells. In EOAL we need only $|E|$ cells for the LABEL array. Because the edge number "j" is inherently built in and can be calculated by

$$j = \text{Integer}\left(\frac{\text{EOAL address} - |V| + 3}{4}\right) \tag{1}$$

In the case we want to associate k data with each edge, only $k|E|$ additional cells are needed (see Fig. 6(b)). EOAL now becomes an array of $|V| + (k+4)|E|$ cells. The first $|V|$ cells still form the VA subarray while the remaining $(k+4)|E|$ cells form the EA subarray. Each consecutive $(k+4)$ cells of EA correspond to the edge e_j and are denoted respectively by

EF_j, EFP_j, ET_j, ETP_j, $d_j^1, d_j^2, \ldots, d_j^k$ where d_j^q denotes the qth data of edge e_j. The formula for finding the edge number "j" for this case becomes

$$j = \text{Integer}\left(\frac{\text{EOAL address} - |V| + 3 + k}{4+k}\right) \tag{2}$$

Let us now briefly discuss how to fetch the stored data in EOAL. Let us assume that we are at one of the end vertices (i.e. either v_{f_j} or v_{t_j}) of edge e_j in EOAL and we want to fetch the qth data of e_j (i.e., d_j^q). We can use the EOAL address of this vertex to find the EOAL address of d_j^q as follows:

$$\text{EOAL address of } d_j^q = |V| + (4+k)(j-1) + q + 4 \tag{3}$$

where j is the edge number obtained through (2). The reason that we have to go through rather lengthy computation in fetching data is that we do not know whether we are at v_{f_j} or v_{t_j}. To overcome this difficulty, we can associate a negative sign with the first data d_j^1 of every edge. Using the positive nature of ET_j, d_j^q can then be fetched efficiently through the flowchart shown in Fig. 9.

Since EOAL basically contains the conventional adjacency list as shown in Fig. 8(g), insertion and/or deletion of any vertex and/or edge for BOAL requires the same operations as that of the conventional adjacency list. However, during the updating of the availability list [1] (i.e., a list of available storage spaces), in the case of deletion of edges, EOAL is faster in the sense that every deleted edge requires one modification in the availability list as compared to two modifications for the conventional adjacency list. Besides, when there are multi-edges in the graph, EOAL is clearly superior in distinguishing one edge from another. Finally, it should be pointed out that, for directed graphs,

EF_j and EFP_j are no longer needed and EOAL simply reduces to the conventional adjacency list.

4. Conclusion

A modification for the conventional adjacency list, called the Edge-Oriented Adjacency List, is introduced for representing undirected graphs. It is shown that EOAL is superior to the conventional adjacency list. Not only does it require less storage space but also it is more efficient in graph modification operations.

References

[1] D. E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Addison-Wesley, 1969. Chap. 2.

[2] D. E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching. Addison-Wesley, 1973.

[3] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," SIAM Journal on Computing, Vol. 1, No. 2, June 1972, pp. 146-160.

[4] R. E. Tarjan, "Finding Minimum Spanning Trees," University of California, Berkeley, Electronics Research Laboratory, Memorandum No. ERL-M501, February 1975.

[5] D. J. Rose and R. E. Tarjan, "Algorithmic Aspects of Vertex Elimination on Graphs," University of California, Berkeley, Electronics Research Laboratory, Memorandum No. ERL-M483 , November 1974.

[6] N. Deo, Graph Theory with Applications to Engineering and Computer Science. Prentice-Hall, 1974. Chap. 14.

[7] S. Crespi-Reghizzi and R. Morpurgo, "A Language for Treating Graphs," Comm. of ACM, Vol. 13, No. 5, May 1970, pp. 319-323.

Figure Captions

Fig. 1 Graph $G = (V,E)$ with $|V| = 4$ and $|E| = 5$

Fig. 2 The conventional adjacency list for G in Fig. 1 with "$*_i$" denoting the end-of-list sign associated with vertex v_i

(a) standard form

(b) modified form utilizing the assumption that vertices are numbered from 1 to $|V|$

Fig. 3 The conventional adjacency list having k data associated with each edge.

Fig. 4 The conventional adjacency list with edge identification

Fig. 5 A modified adjacency list containing both edge data and edge identification

Fig. 6 EOAL structure

(a) without edge data

(b) with edge data

Fig. 7 Flow-chart for constructing BOAL

Fig. 8 EOAL for example in Fig. 1

(a) partial array

(b) after inclusion of edge e_1

(c) after inclusion of edge e_2

(d) after inclusion of edge e_3

(e) after inclusion of edge e_4

(f) after inclusion of edge e_5, this is also the complete EOAL structure

(g) adjacency list extracted from EOAL

Fig. 9 Flow-chart for fetching data

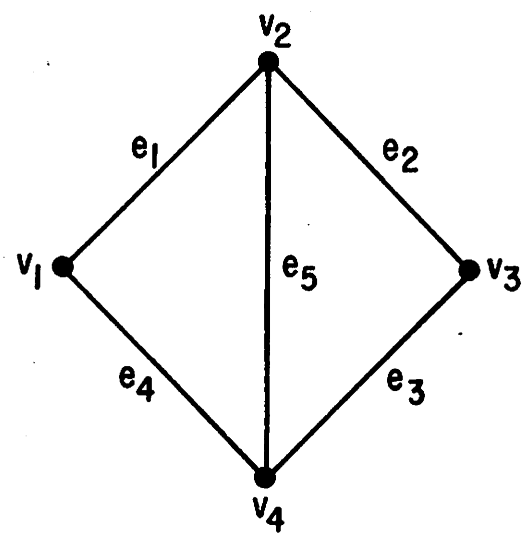
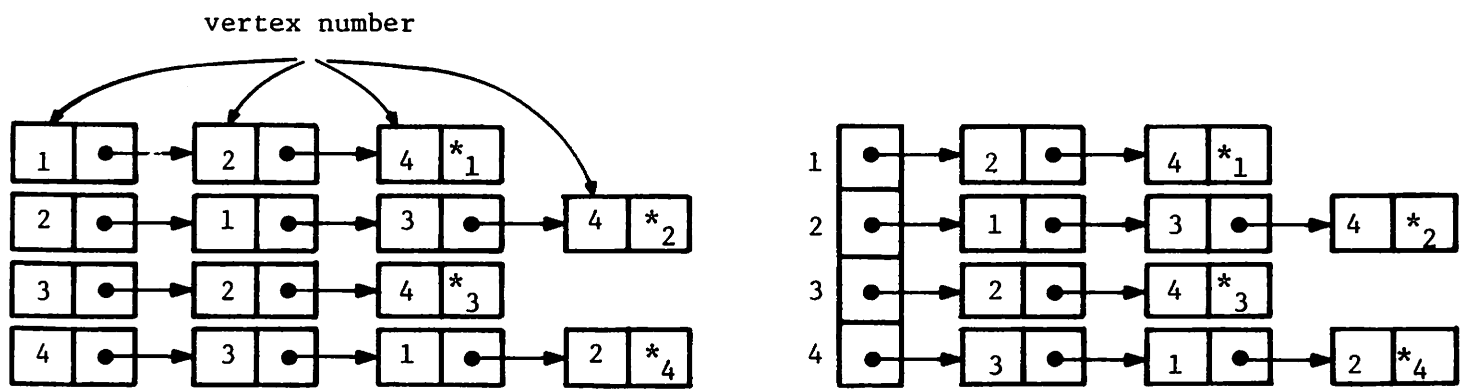


Fig. 1.



(a) (b)

Fig. 2

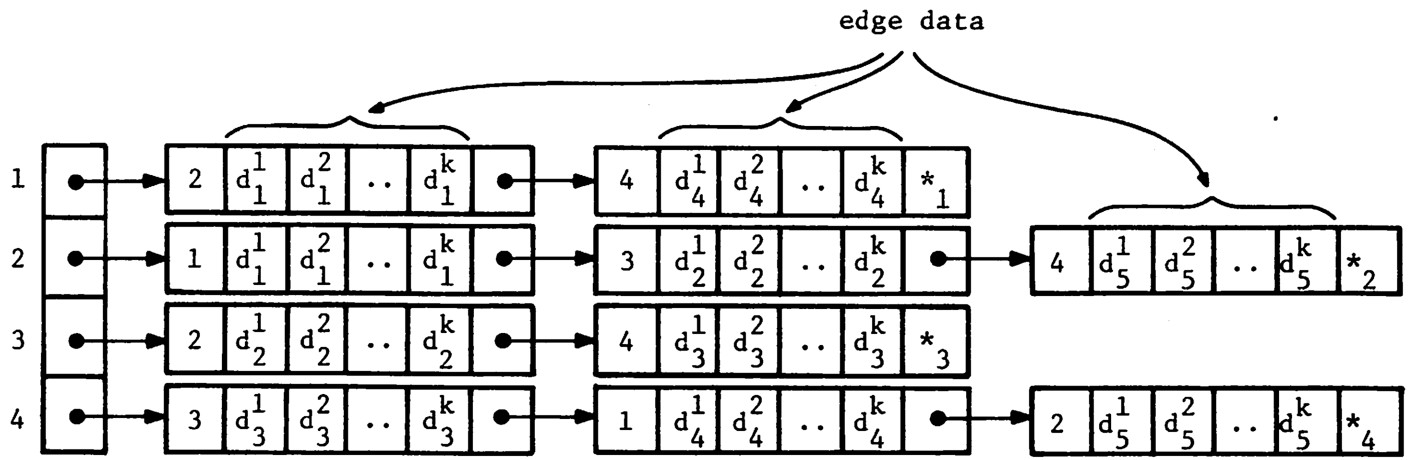


Fig. 3

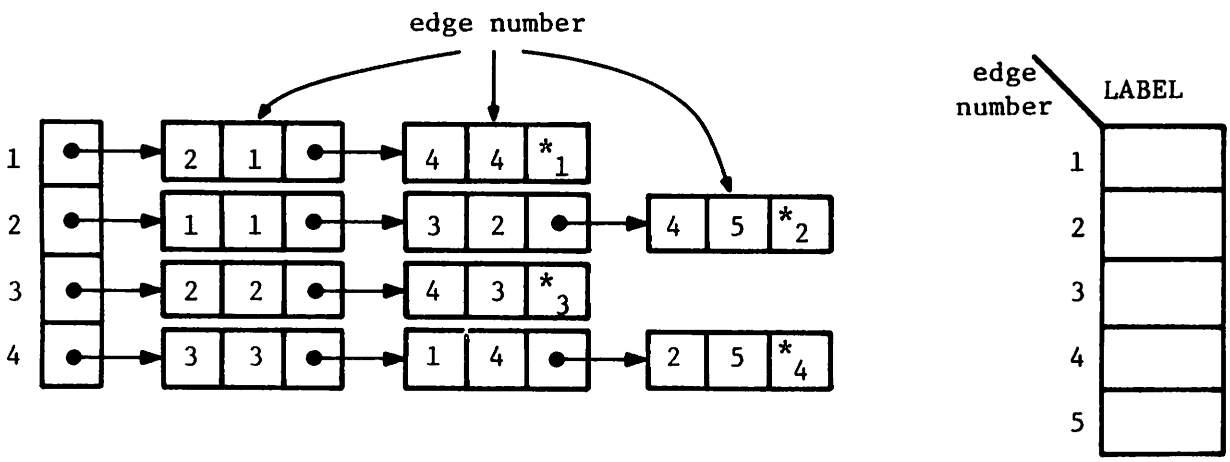
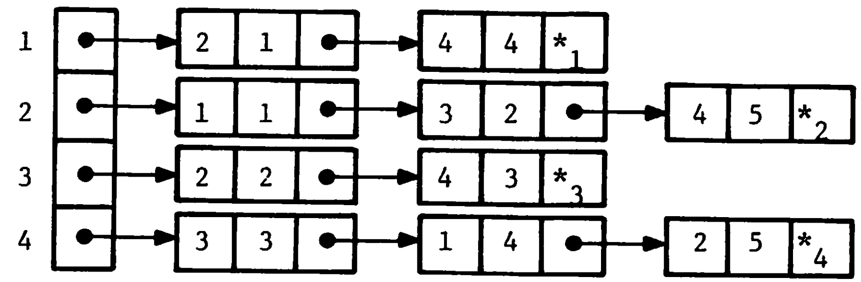

edge number
edge number
LABEL
1
2
3
4
5
2 1
4 4 *1
1 1
3 2
4 5 *2
2 2
4 3 *3
3 3
1 4
2 5 *4
1
2
3
4
5

Fig. 4



edge number	d^1	d^2	..	d^k	LABEL
1	d_1^1	d_1^2	..	d_1^k	
2	d_2^1	d_2^2	..	d_2^k	
3	d_3^1	d_3^2	..	d_3^k	
4	d_4^1	d_4^2	..	d_4^k	
5	d_5^1	d_5^2	..	d_5^k	

Fig. 5

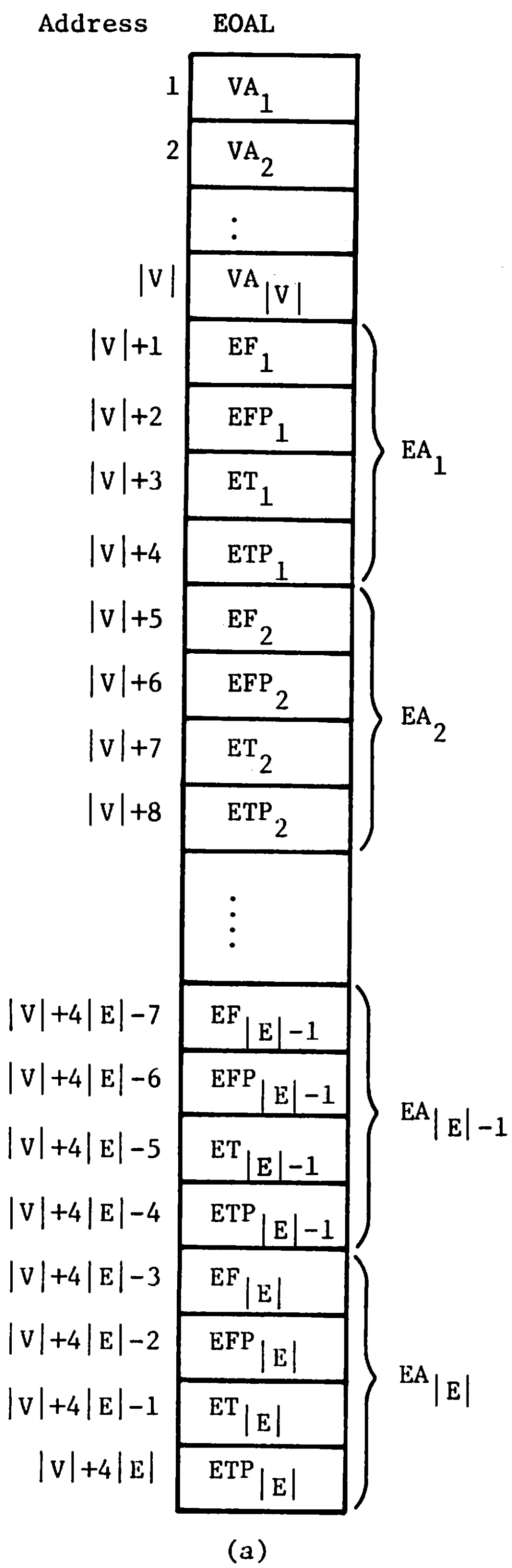

Address
EOAL
1
VA_1
2
VA_2
$|V|$
$VA_{|V|}$
$|V|+1$
EF_1
$|V|+2$
EFP_1
$|V|+3$
ET_1
$|V|+4$
ETP_1
EA_1
$|V|+5$
EF_2
$|V|+6$
EFP_2
$|V|+7$
ET_2
$|V|+8$
ETP_2
EA_2
$|V|+4|E|-7$
$EF_{|E|-1}$
$|V|+4|E|-6$
$EFP_{|E|-1}$
$|V|+4|E|-5$
$ET_{|E|-1}$
$|V|+4|E|-4$
$ETP_{|E|-1}$
$EA_{|E|-1}$
$|V|+4|E|-3$
$EF_{|E|}$
$|V|+4|E|-2$
$EFP_{|E|}$
$|V|+4|E|-1$
$ET_{|E|}$
$|V|+4|E|$
$ETP_{|E|}$
$EA_{|E|}$
(a)
Address
EOAL
1
VA_1
2
VA_2
$|V|$
$VA_{|V|}$
$|V|+1$
EF_1
$|V|+2$
EFP_1
$|V|+3$
ET_1
$|V|+4$
ETP_1
$|V|+5$
d_1^1
$|V|+6$
d_1^2
$|V|+k+4$
d_1^k
EA_1
$|V|+(k+4)(|E|-1)+1$
$EF_{|E|}$
$|V|+(k+4)(|E|-1)+2$
$EFP_{|E|}$
$|V|+(k+4)(|E|-1)+3$
$ET_{|E|}$
$|V|+(k+4)(|E|-1)+4$
$ETP_{|E|}$
$|V|+(k+4)(|E|-1)+5$
$d_{|E|}^1$
$|V|+(k+4)(|E|-1)+6$
$d_{|E|}^2$
$|V|+(k+4)|E|$
$d_{|E|}^k$
$EA_{|E|}$
(b)

Fig. 6

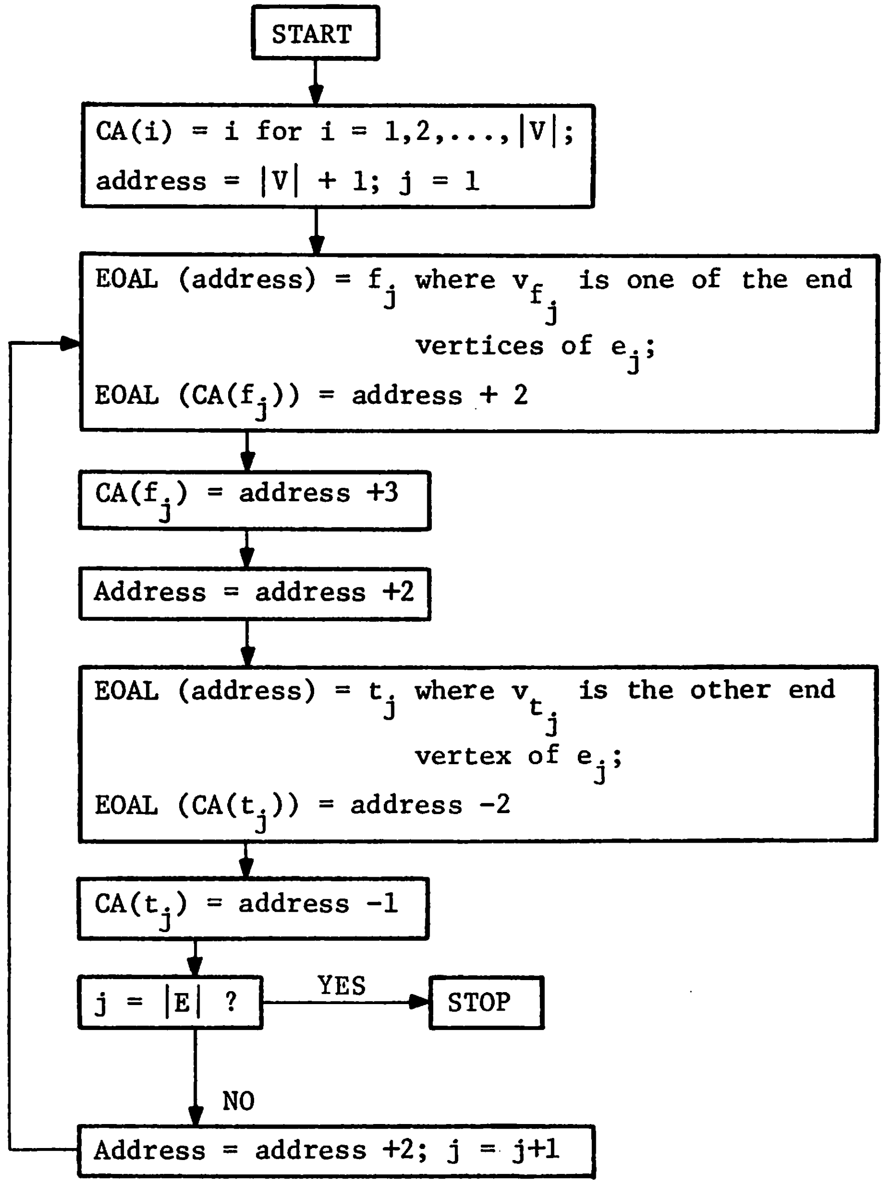

START
CA(i) = i for i = 1,2,...,|V|;
address = |V| + 1; j = 1
EOAL (address) = f_j where v_{f_j} is one of the end vertices of e_j;
EOAL (CA(f_j)) = address + 2
CA(f_j) = address +3
Address = address +2
EOAL (address) = t_j where v_{t_j} is the other end vertex of e_j;
EOAL (CA(t_j)) = address -2
CA(t_j) = address -1
j = |E| ?
YES
STOP
NO
Address = address +2; j = j+1

Fig. 7.

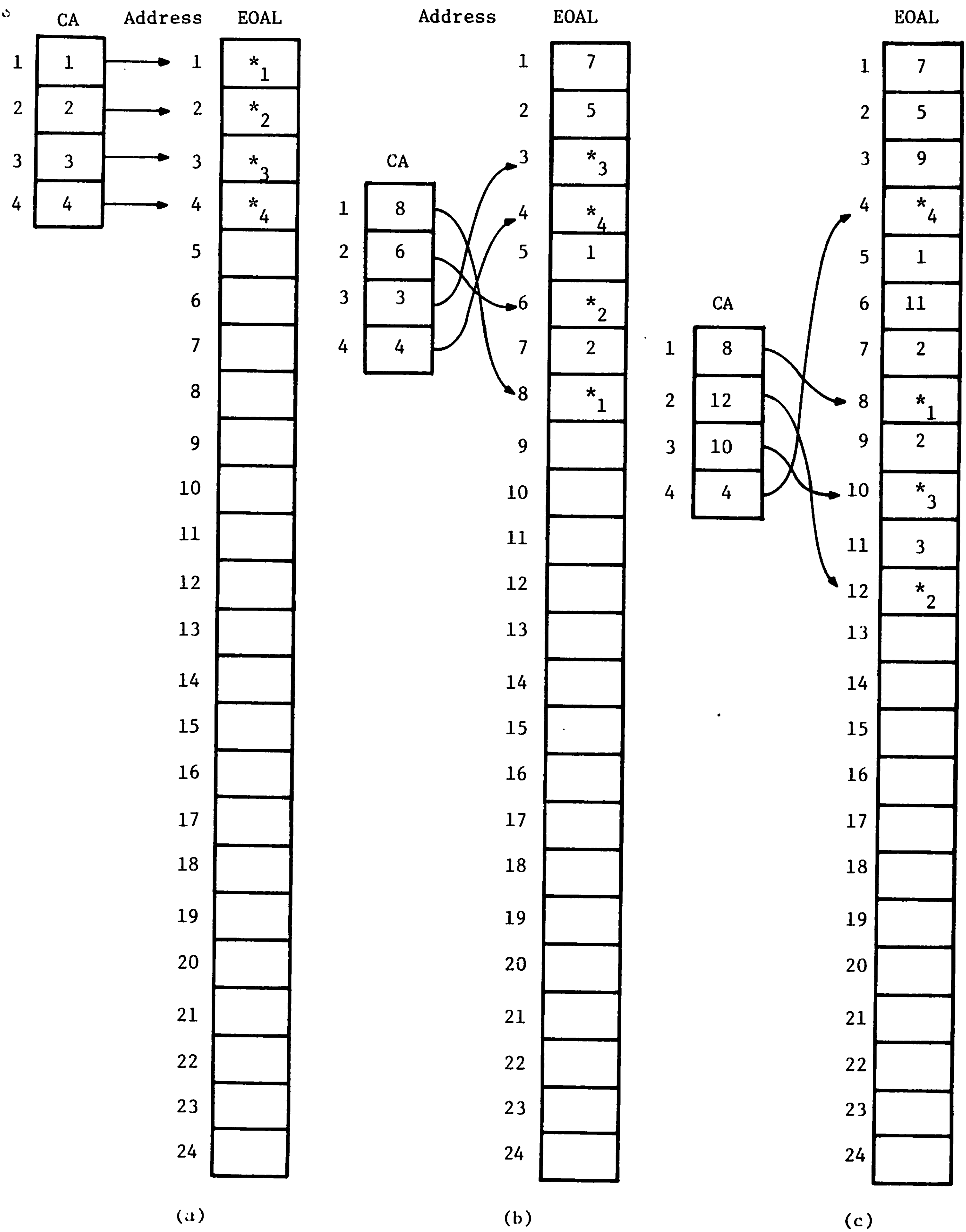

CA
Address
EOAL
*1
*2
*3
*4
(a)
Address
EOAL
CA
7
5
*3
*4
1
*2
2
*1
8
6
3
4
(b)
EOAL
CA
7
5
9
*4
1
11
2
*1
2
*3
3
*2
8
12
10
4
(c)

Fig. 8

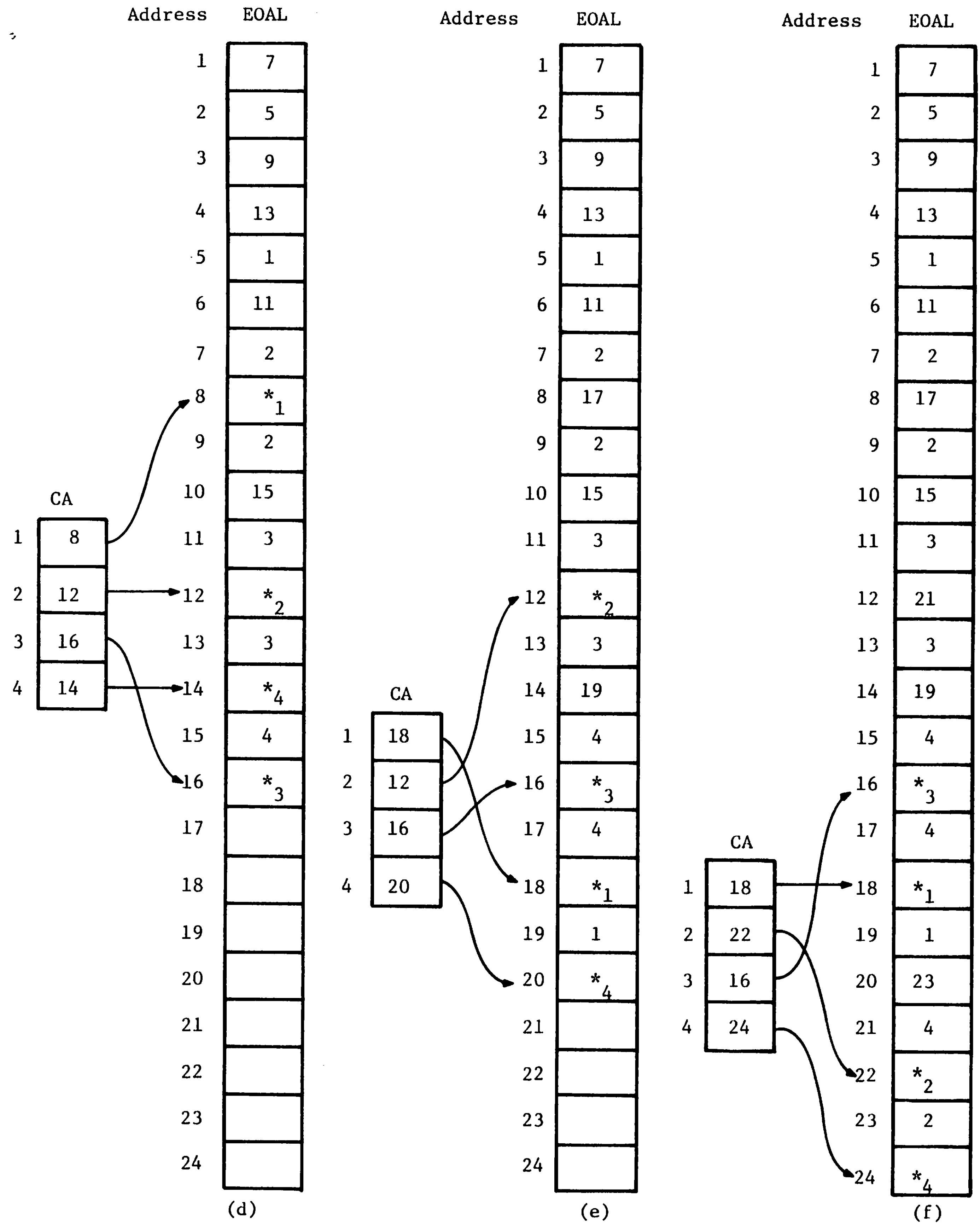


Fig. 8

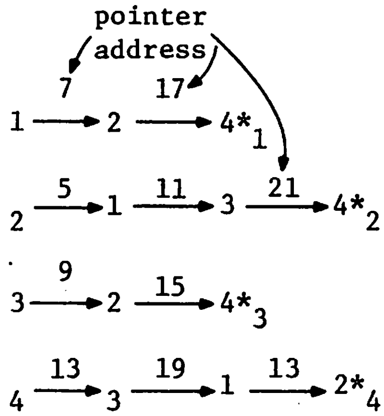

pointer
address
1 → 2 → 4*1 (7, 17)
2 → 1 → 3 → 4*2 (5, 11, 21)
3 → 2 → 4*3 (9, 15)
4 → 3 → 1 → 2*4 (13, 19, 13)

Fig. 8 (g)

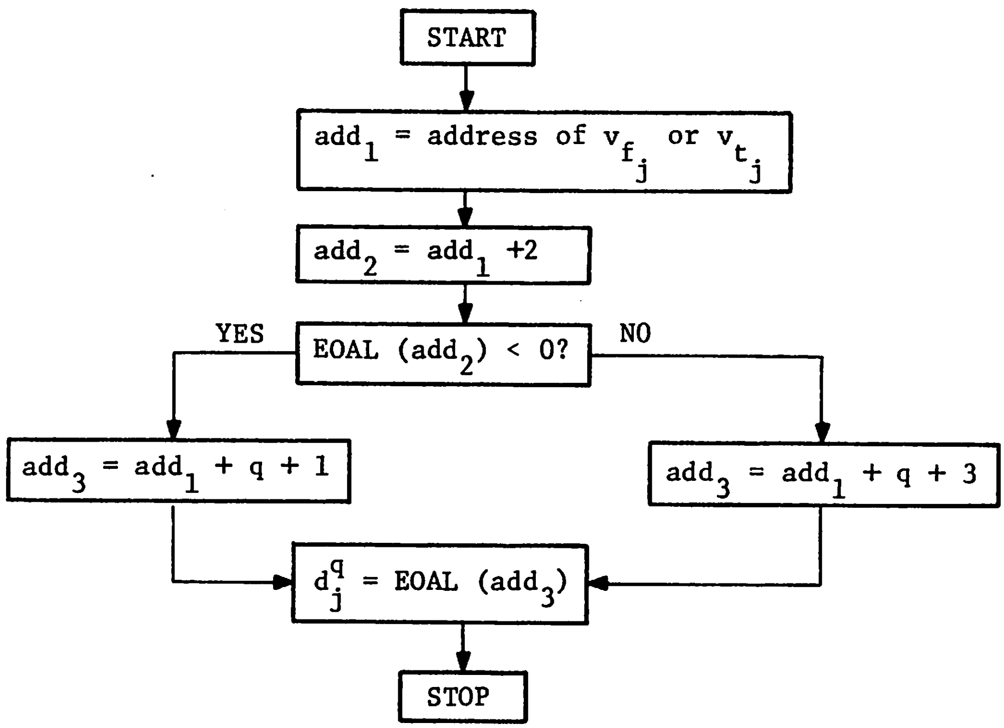

START
add1 = address of vfj or vtj
add2 = add1 +2
EOAL (add2) < 0?
YES
NO
add3 = add1 + q + 1
add3 = add1 + q + 3
dqj = EOAL (add3)
STOP

Fig. 9.