# A COMBINATORIAL ANALYSIS OF BOUNDARY DATA STRUCTURE SCHEMA

T. C. Woo

Department of Industrial & Operations Engineering
University of Michigan
Ann Arbor, Michigan

Technical Report 84-12

April 1984

# A COMBINATORIAL ANALYSIS OF BOUNDARY DATA STRUCTURE SCHEMA

T. C. Woo
Department of Industrial and Operations Engineering
University of Michigan
Ann Arbor, Michigan

April, 1984

Keywords: Data Structure, Data Base, Solid Modeling, Time and Storage Complexities, Optimal Design

## List of Figures

## List of Tables

The best way to design a geometric algorithm is to invoke a powerful insight[1] so that its implementation runs faster than any of the existing algorithms for the same problem. But such a wish does not always come true. There is no guarantee that, for a given problem, one can arrive at an insight of the "Aha" quality[2]. For the sake of argument, suppose one does. Furthermore, suppose the insight does not quite fit the problem. There may be a temptation to alter the original problem to fit a better solution. Short of these two stumbling blocks, the prospect of designing an efficient geometric algorithm can still be fairly discouraging. Consider the time-storage tradeoff in which one faces the "rob-Peter-to-pay-Paul" dilemma.

From a given data structure, one can design an algorithm whose run-time efficiency can be analyzed with techniques in computational complexity[3,4]. To make a given algorithm with a known time complexity run faster, the guaranteed way is to modify the data structure, without changing the problem, by pre-storing the result of some of the intermediate steps which would need to be computed otherwise. Clearly, the net result is a speed-up at the cost of additional storage. The questions are: (i) How much does one gain? and (ii) Can the escalation continue without bound? This paper attempts to answer these two questions in the context of three-dimensional (3D) data structures.

## 1. Introduction

While the design of an "optimal" 3D data structure[5] may be of theoretical interest, its real reward resides in the software speed-up in geometric algorithms for solid modeling, computer aided design, computer aid manufacturing and robotics. Consider solid modeling as a 3D data structure synthesizer whereby a complex solid in some sort of user description is transformed into an internal representation by a set of geometric algorithms that perform, for example, Boolean operations on simpler solids such as cubes and cylinders[6,7]. One can then perform 3D triangulation on the data structure for finite element preprocessing[8,9], ray tracing to extract mass properties[10,11], tool path generation[12,13] algorithms for numerical control, and collision avoidance[14] algorithms for robot path planning[15]. Thus, the speed-up may have an N-fold advantage where N is the number of application algorithms.

There are three major schemes for representing 3D objects[16] -- spatial occupancy of cells in an Octree[17], Boolean combination of solids in a CSG-tree[18], and topological relationship of vertices, edges and faces in a boundary graph[19]. The domain of this paper is the boundary representation.

A curious phenomenon exists in the community of 3D geometric algorithm developers using the boundary representation. While the winged-edge data structure[20] is widely used by solid modeling researchers[16], the theoretical basis of relational topology[21,22]

has not received equal attention. Furthermore, in its twelve years of existence, there has been little analytic rationalization on its time and storage efficiency by users of the winged-edge data structure. A "Catch-22" scenario follows. Indeed, the design of a new data structure may require a powerful geometric insight. The justification of its superiority over the current champion would require analytic measures. Without the measures, it would be difficult to compare data structures convincingly. Without a new challenger, there would be little motivation to develop tools for measuring performance.

It is the objective of this paper to provide techniques for designing new boundary data structures to benefit 3D geometric algorithm developers. Specifically, it shows that:

(i) there is a set of nine data structure accessing and updating primitives common to many 3D geometric algorithms and

(ii) there are over five hundred data structure designs for linking vertices, edges and faces. But,

(iii) there are lower and upper bounds for both the storage requirement and the run-time performance which are established in this paper, and, in particular,

(iv) it is possible to get the most out of run-time performance of a 3D data structure at a fixed storage cost.

## 2. Relations, Combinations and Other Basic Concepts

A boundary data structure can be thought of as a set of relationships among topological entities[23,24]. Let a relation be denoted by

$$X \text{-->} Y$$

where X, Y can be vertices (V), edges (E), faces (F) and holes (H). A relation E-->V, for example, stores the two vertices for each of the edges. Hence, given an edge, its associated vertices can be retrieved or updated.

Consider the number of possible boundary data structure designs. Suppose the topological entities are V, E, and F. (A hole can be implicitly represented by the directions of its edges and its surface normals.) A graph with three nodes and nine arcs is shown in Figure 2.1(a). It is clear that it takes a minimum of two arcs to connect the three nodes. There are

$$C_2^9 = (9! * 8! * 7!) / (2! * 7!) = 36$$

combinations, some of which are not valid because of disconnectedness. It is also possible to store three relations in a data structure. Of the $C_3^9$ or 168 combinations, some are again invalid. In general, there are altogether:

$$C_2^9 + C_3^9 + C_4^9 + C_5^9 + C_6^9 + C_7^9 + C_8^9 + C_9^9 = 502$$

combinations. The winged-edge data structure[20], with an edge pointing to its two vertices, two faces, and four of the possibly many edges, while a face or a vertex points to one of their many edges, is shown in Figure 2.1(b). It is but one of the five hundred or so combinations.

<Insert Figure 2.1>

Having stated the scope of the problem, it is useful to outline the basic concepts for evaluating the storage and time complexities. They are: query on relations and expressing a reverse relation indirectly. The issue is storage versus time.

Using counting formulas discussed in Section 4, each relation can be assigned a storage cost in terms of the total number of edges in the object. Hence, a set of relations represents the "static" view of the data structure with a storage cost. By defining basic queries for accessing and updating, a relation that is not directly stored in a given data structure can be expressed as a procedure in terms of relations that are stored. Hence, the "dynamic" view of a given data structure is presented by the way it is accessed directly or indirectly.

Consider the data structure shown in Figure 2.2(a). It corresponds to one in which a face is linked to all of its edges and an edge is linked to both of its vertices. The dashed arrow in Figure 2.1(b) corresponds to the query of : "Given a face, find all the vertices around it." Clearly, F-->V can be expressed indirectly as F-->E and E-->V. Consider the example in Figure 2.2(c) where the dashed arrow E-->F corresponds to a reverse relation. If a relation V-->F existed in the data structure then E-->F could be computed indirectly as E-->V and V-->F. Otherwise, it would require a "file inversion" to reverse the stored

relation F-->E. Such an operation can take up to order N time while incurring order N intermediate storage, where N is the number of faces F in the stored relation F-->E for this example. The notion of storage-dependent time complexity of a data structure design may be illustrated in another example as shown in Figure 2.2(d). The data structure has two relations E-->V and E-->F. Notice that there is no arc entering E. Answering any of the queries of the type X-->E would require exhaustive search through all vertices or all faces which again is of order N.

<Insert Figure 2.1>

The observations made in the preceeding paragraph will be formalized in the subsequent sections. They serve as the basis for designing and evaluating data structure schema.

## 3. Terminologies

The storage complexity of a data structure is measured by counting formulas and the time complexity of a data structure is measured by a set of primitive queries and update routines. To facilitate the discussion, the following nomenclatures are used.

| | |
|---|---|
| V | : total number of vertices |
| E | : total number of edges |
| F | : total number of faces |
| $V_i$ | : a vertex |
| $E_i$ | : an edge |

$F_i$ : a face

$VV_i$ : number of vertices around a vertex $V_i$

$EV_i$ : number of edges connected to vertex $V_i$

$FV_i$ : number of faces intersecting at $V_i$

$VE_i$ : number of vertices per edge $E_i$

$EE_i$ : number of edges connected to edge $E_i$

$FE_i$ : number of faces intersecting at $E_i$

$VF_i$ : number of vertices around face $F_i$

$EF_i$ : number of edges around face $F_i$

$FF_i$ : number of faces around face $F_i$

It may be noted that the storage complexity of a relation X-->Y can be computed by taking the sum of:

$$\sum_{i}^{X} YX_i$$

where X, Y can be V, E or F and i is summed over all X. For example, the total storage for E-->V is

$$\sum_{i}^{E} VE_i$$

The enumeration of V, E, and F induces nine data structure access primitives AP and update primitives UP.

T1
AP1: Given $V_i$, find all the $VV_i$ vertices connected to it.
UP1: Given $V_i$, link it to all the $VV_i$ vertices.

T2
AP2: Given $V_i$, find all the $EV_i$ edges connected to it.
UP2: Given $V_i$, link it to all the $EV_i$ edges.

AP3: Given $V_i$, find all the $FV_i$ faces around it.

T3

UP3: Given $V_i$, link it to all the $FV_i$ faces.

AP4: Given $E_i$, find all the $VE_i$ vertices connected to it.

T4

UP4: Given $E_i$, link it to all the $VE_i$ vertices.

AP5: Given $E_i$, find all the $EE_i$ edges connected to it.

T5

UP5: Given $E_i$, link it to all the $EE_i$ edges.

AP6: Given $E_i$, find all the $FE_i$ faces intersecting at it.

T6

UP6: Given $E_i$, link it to all the $FE_i$ faces.

AP7: Given $F_i$, find all the $VF_i$ vertices around it.

T7

UP7: Given $F_i$, link it to all the $VF_i$ vertices.

AP8: Given $F_i$, find all the $EF_i$ edges around it.

T8

UP8: Given $F_i$, link it to all the $EF_i$ edges.

AP9: Given $F_i$, find all the $FF_i$ faces around it.

T9

UP9: Given $F_i$, link it to all the $FF_i$ faces.

For convenience, both $AP_i$ and $UP_i$ will be referred to as a topological query $T_i$, for i = 1, 2, ...9. Hence, there are nine such queries T1 - T9, corresponding to the time complexity measures for the nine relations V-->V, V-->E, ... F-->F.

## 4. Storage and Time Complexity

The purpose of this section is two-fold: (i) to introduce the techniques for counting storage cells and for evaluating the time required for answering T1 - T9, and (ii) to establish the lower bound

and the upper bound for both storage and time for all data structures.

It is clear that the eight classes of data structures $C_k^9$, k = 2, 3,... 9, vary by the number of relations stored. Correspondingly, they vary by the time required to answer all T1 - T9. The two extreme classes $C_2^9$ and $C_9^9$ will be studied with the stated dual-purpose in mind.

## 4.1 The $C_2^9$ Class

Consider a $C_2^9$ data structure as shown in Figure 4.1. Implemented as arrays, the storage for the two relations E-->V and E-->F require 2E + 2E = 4E cells. This is because each edge $E_i$ has two vertices, FRONT-V and REAR-V, as well as two faces, LEFT-F and RIGHT-F. As there are E such edges, the total storage is 4E cells.

<Insert Figure 4.1>

The time complexity for the data structure shown in Figure 4.1 can be analyzed as follows. Since the two relations stored are E-->V and E-->F, the two corresponding queries T4 and T6 can be answered in constant time C as the arrays allow direct access. To answer any of the other seven queries, however, a "file inversion" must take place. For example, to answer T2 for V-->E, the following procedure can be written, where $V_i$ is the given vertex and $<E_j>$ is the set of edges connected to $V_j$.

```
Procedure T2 (V_i,  <E_j>)
    E_j<--0
    for n <--1, E do
          for m <--1, 2 do
               if ARRAY(n,m) = V_i then  <E_j> <-- n + <E_j>
          end
               if ARRAY(n,m) = V_i then <E_j> <-- n + <E_j>
     end
end procedure T2
```

Since the outer loop indexed by n is executed E times and the inner loop is executed 2 times, the time complexity for T2 is 2E or O(E). It is not difficult to construct similar procedures and arrive at the summary given in Table 4.1.

<Insert Table 4.1>

## 4.2 The $C_9^9$ Class

If all nine relations are stored, the time complexity for all T1 - T9 is clearly constant. The storage cost for all nine relations are analyzed as follows.

Figure 4.1 shows that the relations E-->V and E-->F cost 2E each, hence leading to the following lemma.

**Lemma 4.1** $\sum_{i}^{V} VE_i = \sum_{i}^{F} FE_i = 2E$

Next, consider the relations V-->E and F-->E. To store a V-->E relation, all the $EV_i$ edges from a vertex $V_i$ must be stored; for all V vertices. Effectively, all the edges are stored exactly twice. Hence, the storage cost for V-->E is 2E. Similarly, the storage cost for F-->E is also 2E. This proves the next Lemma.

**Lemma 4.2** $\sum_{i}^{E} EV_i = \sum_{i}^{F} EF_i = 2E$

The storage cost for relation V-->F is $\sum_{i}^{V} FV_i$. Summed over V, the number of faces per vertex $FV_i$ is exactly the same as summed over all F the number of vertices per face $VF_i$, $\sum_{i}^{F} VF_i$. Similarly, $\sum_{i}^{V} VV_i = \sum_{i}^{F} FF_i$. To evaluate these two pairs of sums, the following lemma is needed.

**Lemma 4.3** $\sum_{i}^{V} FV_i = \sum_{i}^{F} VF_i = 2E, \quad \sum_{i}^{V} VV_i = \sum_{i}^{F} FF_i = 2E$

[Proof] At each vertex $V_i$, the number of vertices $VV_i$, the number of edge $EV_i$ and the number of faces $FV_i$ are identical. By Lemma 4.2,

$$\sum_{i}^{V} VV_i = \sum_{i}^{V} EV_i = \sum_{i}^{V} FV_i = 2E$$

Similarly,

$$\sum_{i}^{F} VF_i = \sum_{i}^{F} EF_i = \sum_{i}^{F} FF_i = 2E$$

As the storage cost for eight of the nine relations are established, the cost for the last relation E-->E is given by the following lemma.

Lemma 4.4 $$\sum_{i}^{E} EE_i = 4E - V$$

[Proof] The relation E-->E stores all the $EE_i$ edges around an edge $E_i$. Since $E_i$ has two vertices $V_i$ and $V_j$, $EE_i$ can be broken into two groups of edges: $EV_i + (EV_j - 1)$. Hence, by Lemma 4.2,

$$\sum_{i}^{E} EE_i = \sum_{i}^{V} EV_i + \sum_{j}^{V} EV_j - 1$$

$$= 2E + 2E - V = 4E - V \qquad \square$$

A summary of the storage cost can now be given as Table 4.2.

<Insert Table 4.2>

Two observations may be made from Table 4.2. First, there are four pairs of symmetric relations about E-->E. Second, all the relations cost 2E except E-->E which costs (4E - V).

As the two extreme classes $C_2^9$ and $C_9^9$ have been analyzed, the lower and the upper bounds for storage and time for all nine classes of data structures may be stated without proof.

Theorem 4.1 For all eight classes of data structures, the lower bound for storage is 4E and the upper bound is (20E - V).

Theorem 4.2 For all eight classes of data structure, the lower bound for time is 9C and the upper bound is (8E + 2C) when all nine queries T1-T9 are interogated.

## 5. Reducing Combinatorial Complexity

To effectively analyze the storage and time complexities of each of the $C_k^9$ data structure designs, where k = 2, 3,...9, two techniques are employed. They are reduction and equivalence. The results in this section provide the basis for reducing $C_k^9$ to $C_n^m$, where 9 > m and k $\geq$ n. (As demonstrated in the following section, $C_4^9$ is reduced to $C_2^7$ which in turn is reduced to $C_2^4$ by invoking the results from this section.) The $C_n^m$ combinations can be further grouped into equivalence classes via symmetry hence yielding a manageable number of designs to evaluate.

Observe that some of the relations involve a variable number of cells for storage. The relation V-->E, for example, requires $EV_i$ cells, where $EV_i$ is the number of edges per vertex $V_i$. In the best case, $EV_i$ = 3 for an object with trihedral vertices.

In the worst case, $EV_i = E/2$ for an n-sided pyramid where the apex has E/2 edges. Designed for the worst case, the data structure for a variable relation is expected to be sparse. By contrast, there are relations that involve a constant number of cells for storage. E-->V, for example, involves exactly two vertices for each edge, i.e., $VE_i = 2$ for both the best and the worst case. Based on this observation, the following lemma establishes the criterion for minimum storage.

Lemma 5.1 Store the relation X-->Y, if the number of cells required is constant for the best and the worst cases.

As there are two relations to which Lemma 5.1 applies, the following theorem permits a reduction in combinatorial complexity.

Theorem 5.1 Of the $C_k^9$ possible designs, only $C_{k-2}^7$ are storage efficient designs, for $k > 2$.

[Proof] By Lemma 5.1, only E-->V and E-->F are constant relations. For $k > 2$, storing these two relations reduces the number of choices from 9 to 7 and k to (k - 2). □

The consequence of Theorem 5.1 is that, for any design $C_k^9$, the two relations E-->V and E-->F must necessarily be a part of the data structure.

Consider the addition of a relation at a fixed cost of 2E and the gain in time for answering T1 - T9. As illustrated in Figure 5.1(a),

the addition of F-->E to a $C_2^9$ design costs 2E in storage but gains a two-fold advantage in answering not only $T_8$ but also $T_5$. As shown in Figure 5.1(b), $T_5$ can be answered indirectly through $T_4$, $T_3$ and $T_8$. Compare this with the addition of a self-loop relation $T_5$. The data structure as shown in Figure 5.1(c) has an additional cost of 2E but does not have an additional gain in query time other than for the relation stored. This example prompts a lemma for the type of relations not to store.

<Insert Figure 5.1>

Lemma 5.2 Avoid storing relations of the type X-->X.

As there are three relations of the type prescribed by Lemma 5.2, V-->V, E-->E, and F-->F, Theorem 5.2 follows immediately.

Theorem 5.2 Of the $C_k^9$ possible designs, only $C_{k-2}^4$ are time efficient designs, for $2 \leq k \leq 6$.

[Proof] A reduction of $C_k^9$ to $C_{k-2}^9$ comes from Theorem 5.1. By Lemma 5.2, there are three self-loop relations among the seven not to choose from. Hence, $C_{k-2}^7$ is reduced to $C_{k-2}^4$. However, if $k \geq 6$, one of the self-loop relations must be used. Hence $2 \leq k \leq 6$. □

Though Lemma 5.2 urges the avoidance of relations of the type X-->X, at least one of the three self-loop relations, V-->V, E-->E, or F-->F, must be used if $k > 6$. In other words, in a $C_8^9$ design, for example, two of the three X-->X type relations must

be stored. It is clear from Table 4.2 as to which one of the three not to store.

Lemma 5.3 Avoid storing E-->E.

## 6. Examples

Though it would be useful to examine all eight classes of data structures $C_k^9$, k = 2, 3, ... 9, two classes are illustrated in this section reflecting the techniques discussed in the preceeding two sections. They are: $C_4^9$ and $C_7^9$.

### 6.1 The Optimal $C_4^9$ Data Structure

As there are four relations among nine to be stored, there can be $C_4^9$ or 126 possibilities. However, by Lemma 5.1, E-->V and E-->F must be stored. By Theorem 5.2, the choice is reduced to $C_2^7$ or 21 possibilities. The intermediate result is illustrated in Figure 6.1(a). By Lemma 5.2, relations of the type V-->V, E-->E, and F-->F should be avoided. This reduces the available choices from seven to four. These four choices are shown as dashed lines in Figure 6.1(b). The six designs, as obtained from $C_2^4$ are shown in Figures 6.1(c1) through (c6). By symmetry, designs in Figure 6.1(c2) and (c5) are equivalent. Similarly, designs in Figure 6.1(c3) and (c4) are equivalent. Dropping the equivalent ones, there are only four to compare. They are shown in Figures 6.1(c1),(c2),(c3), and (c6).

<Insert Figure 6.1>

The storage for the four designs are summarized in Table 6.1.

<Insert Table 6.1>

The time for processing T1 - T9, as summarized in Table 6.2, however, is not entirely the same for the four designs. Design c1 is clearly the fastest in the entire $C_4^9$ class.

<Insert Table 6.2>

## 6.2 The Optimal $C_7^9$ Data Structure

As there are seven relations among nine to be chosen, there can be $C_7^9$ or 36 possibilities. However, four of the seven are already determined by the solution to the $C_4^9$ problem. This leaves five to choose from or $C_3^5$. They are V-->V, V-->F, E-->E, F-->V, and F-->F. By Lemma 5.3, E-->E is not to be chosen as k = 7 < 9. Hence, the possibilities are deduced to $C_3^4$ as shown in Figure 6.2.

<Insert Figure 6.2>

By symmetry, Figures 6.2 (c1) and (c2) are equivalent. Again, by symmetry, Figures 6.2(c3) and (c4) are equivalent. Thus, there are only two designs to compare -- (c1) and (c3).

Using 8E for $C_4^9$ as the base, the storage increase for (c1) due to the relations V-->V, V-->F, and F-->V costs an additional 2E + 2E + 2E. The storage increase for (c3) due to V-->V, F-->V, and F-->F, costs an addition of 6E also. Consequently, the designs in Figure 6.2 have identical storage costs of 14E.

The time complexities as summarized in Table 6.3 shows no significant differences either.

<Insert Table 6.3>

A comparison of $C_4^9$ and $C_7^9$ is now in order. By symmetry, EF is of the same order as EV. The time complexity for $C_4^9$ is, therefore, 4EV + 5C, while that of $C_7^9$ is EV + 8C. Ignoring the constant access time C, $C_7^9$ is approximately four times faster than $C_4^9$ while doubling the storage cost.

<Insert Table 6.4>

## 7. Summary and Conclusion

It is established in this paper that the lower bound for storing a three-dimensional object is 4E and the upper bound is (20E - V), where E is the total number of edges and V the total number of vertices. As the response of a data structure can be measured by the low level topological queries for accessing and updating, the lower bound is constant time while the upper bound is linear time.

Between the lower bound and the upper bound there are over five hundred possible designs arising from the eight combinatorial classes $C_k^9$, k = 2, 3, ... 9, where k is the number of relations stored in a data structure. By observing symmetry and the relationship between time and storage, it is shown that the combinatorial complexity of a data structure design problem can be reduced drastically. Two examples, one for reducing $C_4^9$ to $C_2^4$, the other for reducing $C_7^9$ to $C_3^4$, are used to demonstrate the techniques. An incidental surprise is that by going from $C_2^9$ to $C_4^9$, the storage doubles. But, the response time drops from E, the total number of edges, to EV, the number of edges per vertex. The gain in time is, in general, more than double. The same phenomenon is again illustrated by going from $C_4^9$ to $C_7^9$.

It should be noted that no a priori distribution is placed on the utility of T1 - T9. If such a distribution is available, the techniques shown in this paper can be applied to obtain a constant time data structure.

**Acknowledgement**

The author acknowledges IBM, Data Systems Division, Kingston, New York and the Air Force Office of Scientific Research for their support, S. Baksh and K. Nguyen, University of Michigan, for their analysis of $C_4^9$ $C_7^9$ and Prof. T. Kunii, University of Tokyo, for encouragement.

## References


1. J. L. Bentley, "A Case Study in Applied Algorithm Design", IEEE Computer, Vol. 17, No. 2, February 1984, pp.75-88.

2. M. Gardner, "Aha! Gotcha: Paradoxes to Puzzle and Delight", W. H. Freeman and Co., San Francisco, 1982.

3. A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading Massachusetts, 1974.

4. M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., San Francisco, 1979.

5. T. C. Woo and J. D. Wolter, "A Constant Expected Time, Linear Storage Data Structure for Representing 3D Objects", to appear in IEEE Trans. on Systems Man and Cybernetics, Vol. 14, No. 3, 1984.

6. I. C. Braid, "The Synthesis of Solids Bounded by Many Faces", Comm. ACM, Vol 18, No. 4, April 1975, pp.209-216.

7. H. B. Voelcker and A. A. G. Requicha, "Geometric Modelling of Mechanical Parts and Processes", IEEE Computer, Vol 10, No. 12, December 1977, pp.48-57.

8. B. Wordenweber, "Automatic Mesh Generation of 2 and 3 Dimension Curvilinear Manifolds", University of Cambridge, Computer Laboratory, Tech. Report No. 18, November 1981.

9. T. C. Woo and T. Thomasma, "An Algorithm for Generating Solid Elements in Objects with Holes", Computers and Structures, Vol 18, No. 2, 1984, pp.333-342.

10. R. B. Tilove, "Extending Solid Modeling Systems for Mechanical Design and Kimematic Simulation", IEEE Computer Graphics and

Applications, Vol. 3, No. 3, May 1983, pp.9-19.

11. S. D. Roth, "RAy Casting for Modelling Solids", Computer Graphics and Image Processing, Vol. 18, 1982, pp.109-144.

12. A. R. Grayer, "The Automatic Production of Machined Components Starting from a Stored Geometric Description", in Advances in Computer-Aided Manufacture (D. McPherson, ed.), North-Holland Publishing Company, 1977, pp.137-152.

13. T. C. Woo, "Computer Aided Recognition of Volumetric Designs", in Advances in Computer-Aided Manufacture (D. McPherson, ed.), North-Holland Publishing Co., 1977, pp.121-136.

14. T. Lozano-Perez, "Spatial Planning: A Configuration Space Approach", IEEE Trans. Computers, Vol C-32, No. 2, February 1983, pp.108-120.

15. M. A. Wesley, T. Lozano-Perez, L. T. Lieberman, M. A. Lavin, and D. D. Grossman, "A Geometric Modelling System for Automated Mechanical Assembly", IBM Journal of Research and Development, Vol. 24. No. 1, January 1980, pp. 64-74.

16. A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods and Systems", ACM Computing Surveys, Vol. 12, No. 4, December 1980, pp.437-464.

17. D. Meagher, "Geometric Modeling Using Octree Encoding", Computer Graphics and Image Processing, Vol 19, 1982, pp.129-147.

18. A. A. G. Requicha and H. B. Voelcker, "Constructive Solid Geometry", University of Rochester, Production Automation Project, Tech. Memo 25, November 1977.

19. I. C. Braid, "Six Systems for Shape Design and Representation", University of Cambridge, CAD Group Document No. 87, May 1975.

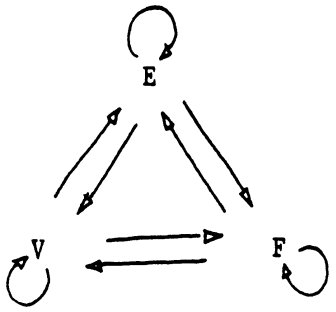
20. B. G. Baumgart, "Winged-edge Polyhedron Representation", Stanford University, Computer Science Department, Report No. CS-320, October 1972.

21. K. Weiler, "Adjacency Relationships in Boundary Graph Based Solid Models", General Electric Corp. Research and Development, Schenectady, New York, June 15, 1983.

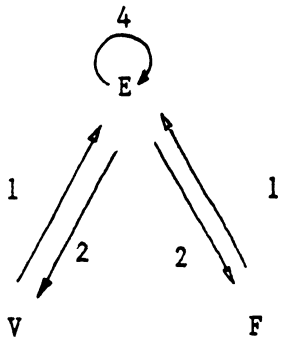
22. P. Hanrahan, "An Introduction to Relational Topology", New York Institute of Technology, Computer Graphics Laboratory, October 1983.

23. A. Baer, C. Eastman, and M. Henrion, "Geometric Modelling: A Survey", Computer-Aided Design, Vol. II, No. 5, September 1979, pp.253-272.

24. I. C. Braid, "On Storing and Changing Shape Information", Computer Graphics, Vol. 12, No. 3, August 1978, pp.252-256.




(a) Nine and three entities




(b) Winged-edge data structure

Figure 2.1 Schema for Boundary Data Structures

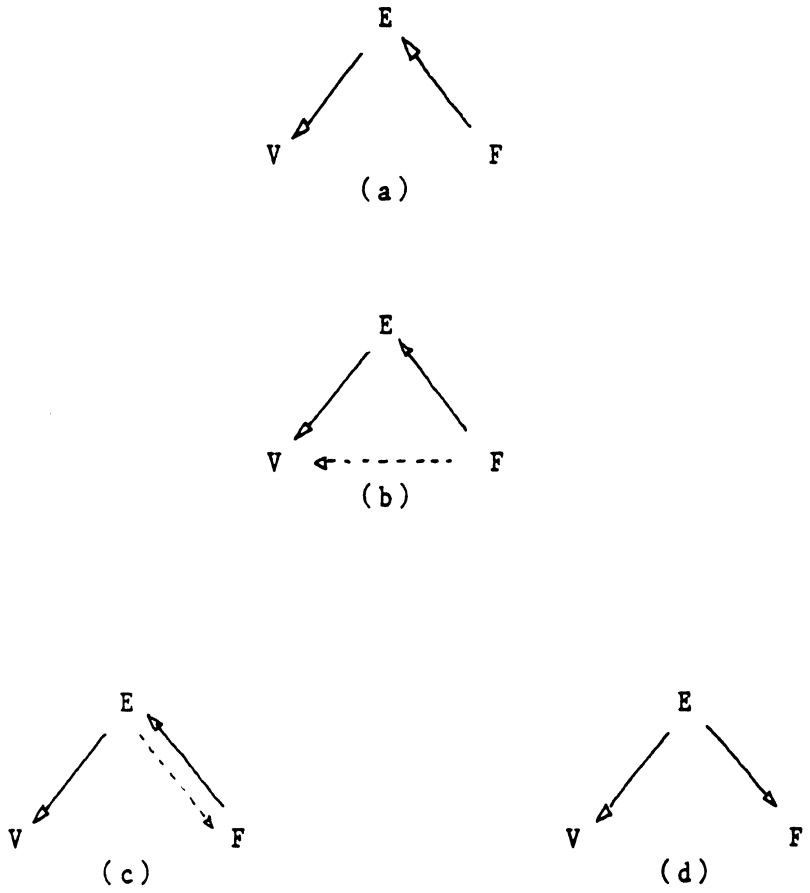
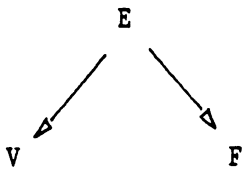



Figure 2.2 Indirect and Reverse Relations




| | FRONT-V | REAR-V | LEFT-F | RIGHT-F |
|---|---|---|---|---|
| $EDGE_1$ | | | | |
| $EDGE_2$ | | | | |
| . | | | | |
| . | | | | |
| . | | | | |
| $EDGE_E$ | | | | |

Figure 4.1 A $C_2^9$ Data Structure Design and Implementation

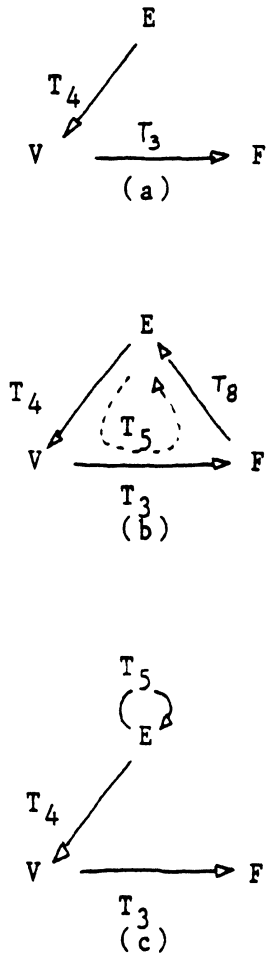
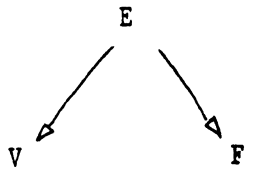
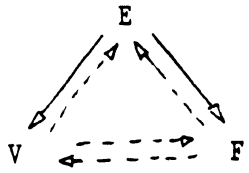


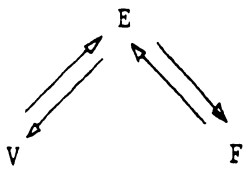

Figure 5.1 Additional time efficiency from fixed storage cost



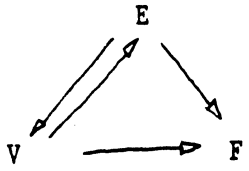

(a)



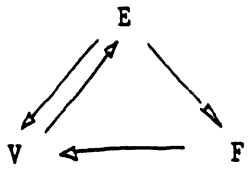

(b)



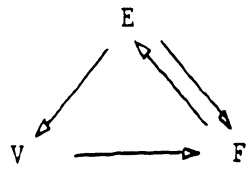

(c1)



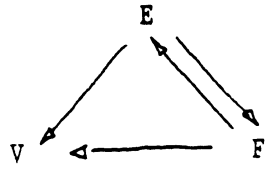

(c2)



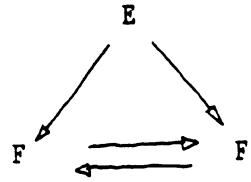

(c3)




(c4)




(c5)




(c6)

Figure 6.1 $C_4^9$ Data Structures

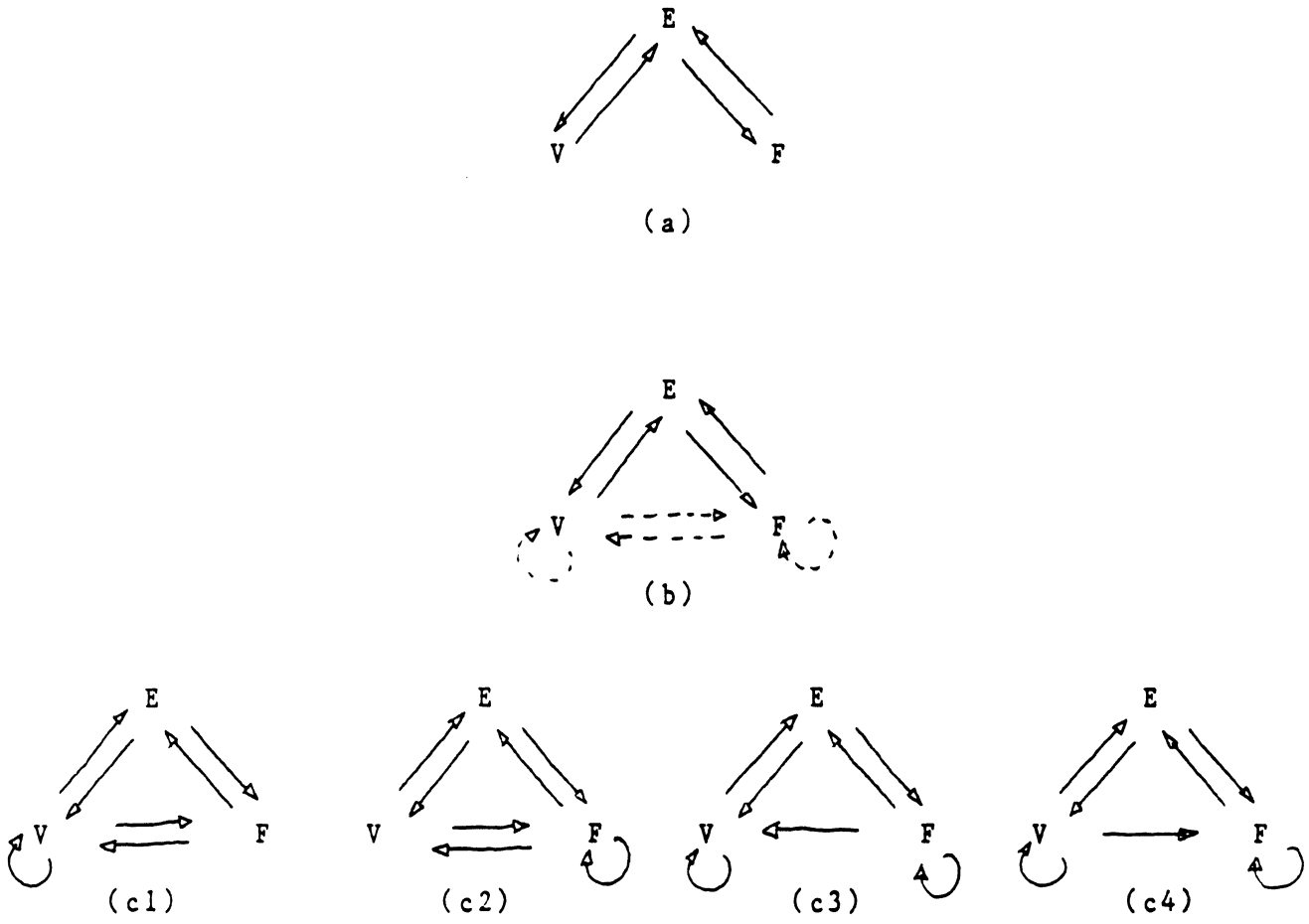



Figure 6.2 $C_7^9$ Data Structures

| T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 |
|---|---|---|---|---|---|---|---|---|
| E | E | E | C | E | C | E | E | E |

where C: constant time

E: time linear in E, in the worst case

Table 4.1 Time Complexity for $C_2^9$ Data Structure

| RELATION | V-->V | V-->E | V-->F | E-->V | E-->E | E-->F | F-->V | F-->E | F-->F |
|---|---|---|---|---|---|---|---|---|---|
| SUMMATION | $\sum_{i}^{V} VV_i$ | $\sum_{i}^{V} EV_i$ | $\sum_{i}^{V} FV_i$ | $\sum_{i}^{E} VE_i$ | $\sum_{i}^{E} EE_i$ | $\sum_{i}^{E} FE_i$ | $\sum_{i}^{F} VF_i$ | $\sum_{i}^{F} EF_i$ | $\sum_{i}^{F} FF_i$ |
| STORAGE | 2E | 2E | 2E | 2E | 4E-V | 2E | 2E | 2E | 2E |

Table 4.2 Storage Complexity of the Nine Relations

| | V-->V | V-->E | V-->F | E-->V | E-->E | E-->F | F-->V | F-->E | F-->F | TOTAL |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 2E | 2E | 2E | | 2E | 2E | | 2E | |
| c1 | | * | | * | | * | | | * | 8E |
| c2 | | * | * | * | | * | | | | 8E |
| c3 | | * | | * | | * | * | | | 8E |
| c6 | | | * | * | | * | * | | | 8E |

Table 6.1 Storage for $C_4^9$ Data Structures

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | TOTAL TIME |
|---|---|---|---|---|---|---|---|---|---|---|
| C1 | EV | C | EV | C | C | C | EF | C | EF | 2EV + 2EF + 5C |
| C2 | EV | C | C | C | C | C | E | E | E | 3E + EV + 5C |
| C3 | EV | C | EV | C | C | C | C | VF | E | E + 2EV + VF + 5C |
| C6 | E | E | C | C | E | C | C | E | E | 5E + 4C |

Table 6.2 Time for $C_4^9$ Data Structures

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | TOTAL TIME |
|---|---|---|---|---|---|---|---|---|---|---|
| c1 | C | C | C | C | C | C | C | C | EF | EF + 8C |
| c3 | C | C | EV | C | C | C | C | C | C | EV + 8C |

Table 6.3 Time for $C_7^9$ Data Structures

| | STORAGE | TIME |
|---|---|---|
| $C_4^9$ | 8E | 4EV + 5C |
| $C_7^9$ | 16E | EV + 8C |

Tables 6.4 Comparison of $C_4^9$ and $C_7^9$ Data Structures