Speed up the software in geometric algorithms for solid modeling, CAD/CAM, and robotics applications. How? By using boundary data structures that are fast and use less storage.

A Combinatorial Analysis of Boundary Data Structure Schemata

Tony C. Woo
University of Michigan

While the design of an efficient 3-D data structure may be of theoretical interest, its real reward resides in the software speedup in geometric algorithms for solid modeling, computer-aided design, computer-aided manufacturing, and robotics. Consider a solid modeler as a 3-D data structure synthesizer. It transforms the user description of a complex solid into an internal representation with a set of geometric algorithms performing Boolean operations on simpler solids such as cubes and cylinders.[1,2] One can then perform 3-D triangulation on the data structure for finite element preprocessing,[3,4] ray tracing to extract mass properties,[5,6] tool-path generation for numerical control,[7,8] and collision-avoidance[9] for robot path planning.[10] Thus, a speedup in the data structure would have an N-fold advantage, where N is the number of application algorithms.

We have three major schemes for representing 3-D objects:[11] spatial occupancy of cells in an octree,[12] Boolean

0272-1716/85/0300-0019$01.00

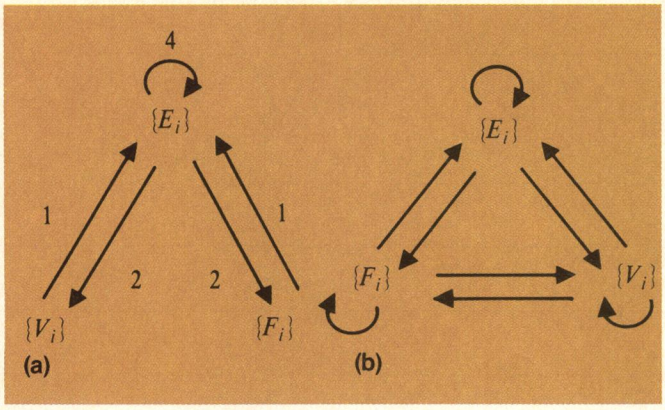


Figure 1. Schemata for boundary data structure: winged-edge data structure (a); and nine relations and three entities (b).

combination of solids in a CSG tree,[13] and topological relationship of vertices, edges, and faces in a boundary graph.[14] The domain of this article is the boundary representation.

A curious phenomenon exists in the community of 3-D geometric algorithm developers using the boundary representation. While the winged-edge data structure[15] is used widely by solid modeling researchers, in its 12 years of existence there has been little analytic rationalization of its time and storage efficiency by its users. A "Catch 22" situation follows. The design of a better data structure may require a powerful geometric insight of the "Aha!" sort.[16,17] The justification of its superiority over the current champion would require analytic measures.[18,19] Without the measures, it would be difficult to compare data structures convincingly. But, without a new challenger, little motivation would exist to develop tools for measuring 3-D data structure performance.

It is the objective of this article to provide techniques for designing new boundary data structures to benefit 3-D geometric algorithm developers. Specifically, I show the following:

(1) There is a set of nine data structure accessing and updating primitives common to many 3-D geometric algorithms.

(2) There are over 500 data structure schemata forming eight storage classes.

(3) But, not all schemata in the same storage class perform the same in time. For a given storage class the comparison of schemata and the construction of an optimal one are illustrated.

(4) In addition, schemata for the eight classes form a storage-time curve in the shape of the letter "L."

(5) The elbow in the L-curve is related to a new data structure presented here. It has a lower storage and a faster time when compared to the winged-edge data structure.

(6) Furthermore, I show that the new data structure is optimal within its storage class.

Relations, combinations, storage, and time

The *storage complexity* of a data structure is measured using counting formulas, and the *time complexity* of a data structure is measured using a set of primitive queries and update routines. To facilitate discussion, I use the following symbols:

V_i	a vertex
E_i	an edge
F_i	a face
$\{V_i\}$	a set of vertices
$\{E_i\}$	a set of edges
$\{F_i\}$	a set of faces
V	total number of vertices
E	total number of edges
F	total number of faces
VV_i	number of vertices connected to V_i, variable
EV_i	number of edges connected to V_i, variable
FV_i	number of faces sharing V_i, variable
VE_i	number of vertices sharing E_i, constant, 2
EE_i	number of edges sharing the VE_i vertices and the FE_i faces of E_i, constant, 4
FE_i	number of faces sharing E_i, constant, 2
VF_i	number of vertices sharing face F_i, variable
EF_i	number of edges around face F_i, variable
FF_i	number of faces around face F_i, variable.

Note the duality between a vertex and a face. In particular, note that the EE_i wings of an edge E_i are self-dual.

A boundary data structure can be thought of as a set of adjacency relationships among topological entities.[20,21] Let a relation be denoted by

$$X \rightarrow Y$$

where X, Y can be a set of vertices $\{V_i\}$, edges $\{E_i\}$, faces $\{F_i\}$, or holes $\{H_i\}$. A relation $\{E_i\} \rightarrow \{V_i\}$, for example, stores two vertices for each of the edges. Hence, given an edge E_i, its associated vertices $\{V_i\}$ can be retrieved or updated. Figure 1a shows the schema for the winged-edge data structure,[15] 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.

Consider the number of possible boundary data structure designs. Suppose the topological entities are V_i, E_i, and F_i. (A hole can be implicitly represented by the directions of its edges and its surface normals or explicitly with cutting edges and faces bridging the boundaries.) With three nodes and nine arcs a graph can be formed as shown in Figure 1b. Clearly, it takes a minimum of two relations to connect the three entities. But, there are $C_2^9 = 36$ such combinations, though some are not valid because of disconnectedness. It is also possible to store three relations in a data structure. Of the C_3^9, or 84, combinations, some are invalid because of disconnectedness. Of the remaining valid ones, some are more efficient than others. If it seems onerous to enumerate the 84 combinations for C_3^9, the task is actually much worse. In general, there are

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

combinations all together.

Having stated the size of the problem, it is useful to outline the basic concepts for evaluating the storage and time complexities. They are queries, relations, indirect relations, and reverse relation.

Using counting formulae discussed in the next section, we can assign each relation a storage cost in terms of the total number of edges E in the object. Therefore, a set of relations with a storage cost represents a static view of the data structure. By defining basic queries for accessing and updating, we can describe a relation that is not stored directly in a given data structure and that can be expressed as a procedure in terms of relations that are stored. Hence, we represent a dynamic view of a given data structure by the way it is accessed indirectly or reversely.

Consider the data structure schema shown in Figure 2a. A face F_i is linked to all of its edges $\{E_i\}$ by $\{F_i\} \rightarrow \{E_i\}$, and an edge E_i is linked to both of its vertices $\{V_i\}$ by $\{E_i\} \rightarrow \{V_i\}$. The dashed arrow in Figure 2b corresponds to the query: "Given a face, find all the vertices around it." Clearly, $\{F_i\} \rightarrow \{V_i\}$ can be expressed *indirectly* as $\{F_i\} \rightarrow \{E_i\}$ and $\{E_i\} \rightarrow \{V_i\}$. Now consider the example in Figure 2c where the dashed arrow $\{E_i\} \rightarrow \{F_i\}$ corresponds to an *inverse* relation. If a relation $\{V_i\} \rightarrow \{F_i\}$ existed in the data structure, $\{E_i\} \rightarrow \{F_i\}$ could be computed indirectly as $\{E_i\} \rightarrow \{V_i\}$ and $\{V_i\} \rightarrow \{F_i\}$. Otherwise it would require a file inversion to invert the stored relation $\{F_i\} \rightarrow \{E_i\}$. Such an operation can take up to order F time, where F is the number of faces in the stored relation $\{F_i\} \rightarrow \{E_i\}$.

Unless ambiguity arises, subscripts and brackets for a relation will be dropped for simplicity. For instance, $\{V_i\} \rightarrow \{F_i\}$ is given as $V \rightarrow F$.

The storage complexity of a relation $X \rightarrow Y$ can be computed by taking the sum

$$\sum_{i=1}^{X} YX_i$$

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

$$\sum_{i=1}^{E} VE_i$$

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

$T1$ $AP1$: 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.

$T3$ $AP3$: Given V_i, find all the FV_i faces around it.
$UP3$: Given V_i, link it to all the FV_i faces.

$T4$ $AP4$: Given E_i, find the two vertices connected to it.
$UP4$: Given E_i, link it to the two vertices.

$T5$ $AP5$: Given E_i, find the four edges connected to it.
$UP5$: Given E_i, link it to the four edges.

$T6$ $AP6$: Given E_i, find the two faces intersecting at it.
$UP6$: Given E_i, link it to the two faces.

$T7$ $AP7$: Given F_i, find all the VF_i vertices around it.
$UP7$: Given F_i, link it to all the VF_i vertices.

$T8$ $AP8$: Given F_i, find all the EF_i edges around it.
$UP8$: Given F_i, link it to all the EF_i edges.

$T9$ $AP9$: Given F_i, find all the FF_i faces around it.
$UP9$: 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 Ti, for $i = 1, 2, \ldots 9$. Hence, we have nine such queries $T1$ through $T9$, corresponding to the time-complexity measures for the nine direct, indirect, or inverse relations $V \rightarrow V$, $V \rightarrow E, \ldots F \rightarrow F$.

Bounds for storage and time complexities

This section introduces the techniques for counting storage and for evaluating the time required for answering $T1$-$T9$ queries and establishes 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_m^9, $m = 2, 3, \ldots 9$ vary by the number of relations stored. Correspondingly, they vary by the time required to answer all $T1$-$T9$ queries. We will study the two extreme classes C_2^9 and C_9^9 with the previously stated twofold purpose in mind.

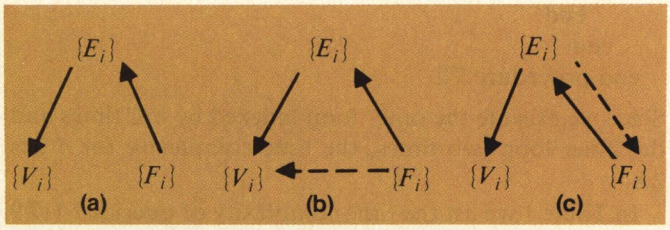


Figure 2. Indirect and inverse relations.

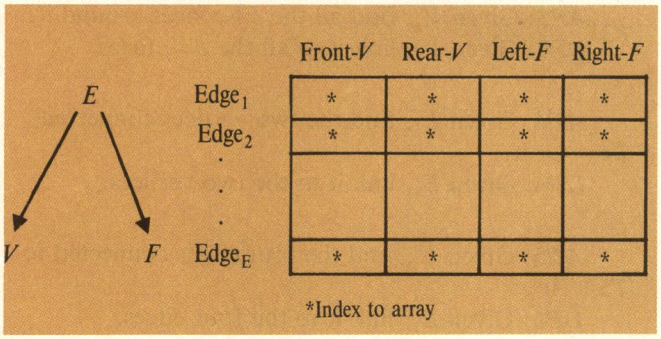


Figure 3. A C_2^9 data structure design and implementation.

Table 1.
Time complexity for C_2^9 data structure.

Queries	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$	$T8$	$T9$
Edges	E	E	E	k	E	k	E	E	E
Total									$7E+2k$

Where k: constant time for direct access and
E: time linear in E, in the worst case

The C_2^9 class. Consider a C_2^9 data structure as shown in Figure 3. Implemented as arrays, the storage for the two relations $E \rightarrow V$ and $E \rightarrow F$ 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 $4E$.

The time complexity for the data structure shown in Figure 3 can be analyzed as follows. Since the two direct relations are $E \rightarrow V$ and $E \rightarrow F$, the two corresponding queries $T4$ and $T6$ can be answered in constant time k as the arrays allow direct access. To answer any of the other seven queries, however, we must force a "file inversion." For example, to answer $T2$ for $V \rightarrow E$, we can write the following procedure where V_i is the given vertex, $\{E_i\}$ is the set of edges connected to V_i, and n is the row index.

```
Procedure T2 (V_i, {E_i})
  {E_i} ← 0
  for n ← 1, E do
    for m ← 1, 2 do
      if ARRAY(n,m) = V_i then {E_i} ← n + {E_i}
    end
  end
end procedure T2
```

Since we execute the outer loop indexed by n E times and the inner loop two times, the time complexity for $T2$ is $2E$.

In Table 1 we list the time complexity of queries $T1$-$T9$ in $O(E)$ by dropping the coefficient, which is dependent on implementation. For instance, the time for $T2$ is given as E, although it is actually $2E$ when it is evaluated against the C_2^9. The total time for all $T1$-$T9$ is $7E+2k$, with the implication that, of the nine queries, seven are linear time and two are constant time.

It can be shown, using the technique illustrated in the Appendix, that all valid schemata in the C_2^9class cost $4E$ in storage and run in $7E+2k$ time. We will use the data structure illustrated in Figure 3 as the base leading to other C_m^9 schemata to be presented in the following sections.

The C_9^9 class. If all nine relations are stored, the time complexity for all $T1$-$T9$ is clearly constant $9k$. We analyze the storage cost.

Figure 3 shows that the relations $E \rightarrow V$ and $E \rightarrow F$ cost $2E$ each, hence leading to the following lemma.

Lemma 1

$$\sum_i^V VE_i = \sum_i^F FE_i = 2E$$

Next, consider the relations $V \rightarrow E$ and $F \rightarrow E$. To store a $V \rightarrow E$ relation, we must store all the EV_i edges from a vertex V_i for all V vertices. But we store all the edges exactly twice. Hence, the storage cost for $V \rightarrow E$ is $2E$. Similarly, the storage cost for $F \rightarrow E$ is also $2E$. This proves the next lemma.

Lemma 2

$$\sum_i^E EV_i = \sum_i^F EF_i = 2E$$

The storage cost for relation $V \rightarrow F$ 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, we need the following lemma.

Lemma 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 edges EV_i, and the number of faces FV_i

Table 2.
Storage complexity of the nine relations.

Relation	$V \rightarrow V$	$V \rightarrow E$	$V \rightarrow F$	$E \rightarrow V$	$E \rightarrow E$	$E \rightarrow F$	$F \rightarrow V$	$F \rightarrow E$	$F \rightarrow F$
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	$2E$	$2E$	$2E$	$2E$	$4E$	$2E$	$2E$	$2E$	$2E$

are identical. By Lemma 2, we get

$$\sum_i^V VV_i = \sum_i^V EV_i = \sum_i^V FV_i = 2E$$

Similarly,

$$\sum_i^F VF_i = \sum_i^F EF_i = \sum_i^F FF_i = 2E$$

As each edge has four wings, the storage cost for relation $E \rightarrow E$ is clearly $4E$.

Lemma 4

$$\sum_i^E EE_i = 4E$$

Table 2 presents a summary of the storage cost.

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 eight classes of data structures may be stated.

Theorem 1

For all eight classes of data structure schemata, the lower bound for *storage* is linear ($4E$), where E is the total number of edges, and the upper bound is also linear ($20E$).

Theorem 2

For all eight classes of data structure, the lower bound for *time* is constant ($9k$), where k is the constant time for direct access, and the upper bound is linear ($7E+2k$) when all nine queries $T1$-$T9$ are interrogated.

Note the relationship in the storage and time trade-off. At a storage cost of $4E$, the time for C_2^9 is $7E+2k$. At a storage cost of $20E$, C_9^9 offers constant time at $9k$. We may hypothesize that a linear relationship exists in that, as the storage goes up from $4E$ to $20E$, the time goes down as $7E+2k$, $6E+3k, \ldots, E+8k$, $9k$, for C_2^9, $C_5^9, \ldots C_8^9$, and C_9^9. In the remaining sections we show that such is *not* the case. Here, we give the intuition.

From Table 2 we see that the storage for each of the nine relations costs $2E$, except for $E \rightarrow E$, which costs $4E$. Hence, the storage for classes C_2^9, C_3^9, . . . C_9^9 does not go up uniformly by $2E$. In other words, the storage progression could be $4E$, $8E$, $10E$, $12E$, . . . $20E$. Or, it could be $4E$, $6E$, $10E$, $12E$, . . . $20E$, or any such progression. The point at which the "wrinkle" occurs, however, should be interesting.

In the next section we correspond the wrinkle in a progression to the winged-edge data structure. It dramatically decreases the time from $7E+2k$ (for C_2^9) to $6EV_i + 3k$, where E is the total number of edges while EV_i is the number of edges at a particular vertex V_i. Hence, we postulate an L-shaped curve as shown in Figure 4. The elbow of the L-curve is below time linear in E, since $EV_i << E$. In a later section we establish the elbow in the L-curve (by introducing a new wrinkle in the progression). The corresponding data structure is shown to be faster ($5EV_i + 4k$) than the winged edge, yet it costs less in storage.

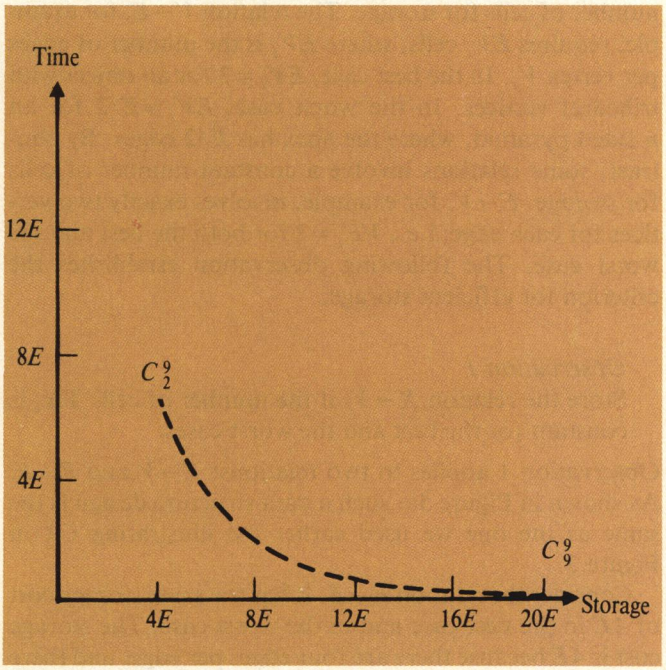


Figure 4. The L-curve.

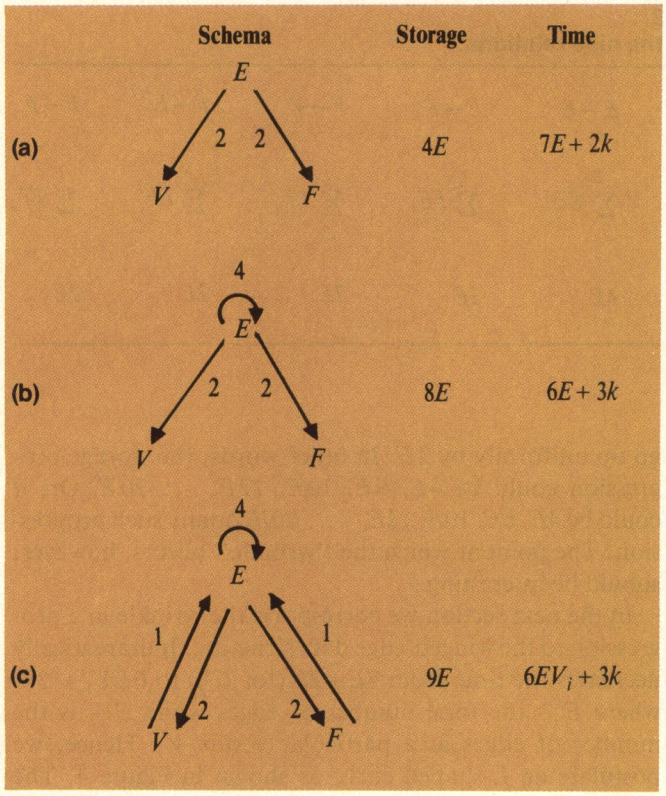


Figure 5. Rediscovering the winged-edge: a C_2^9 schema (a); a C_3^9 schema (b); and the winged-edge schema (c).

Rediscovering the winged edge

Observe that some of the relations involve a variable number of cells for storage. The relation $V \rightarrow E$, 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. By contrast, some relations involve a constant number of cells for storage. $E \rightarrow V$, for example, involves exactly two vertices for each edge; i.e., $VE_i = 2$ for both the best and the worst case. The following observation establishes the criterion for efficient storage.

Observation 1
Store the relation $X \rightarrow Y$, if the number of cells YX_i is constant for the best and the worst cases.

Observation 1 applies to two relations: $E \rightarrow V$ and $E \rightarrow F$. As shown in Figure 5a, such a data structure design is the same as the one we used earlier for illustrating C_2^9 in Figure 3.

Consider the relation $E \rightarrow E$. It has the same storage cost of $4E$ in the best case and in the worst case. The storage cost is $4E$ because there are four edges per edge, and there are E of them.

Observation 2
Store the relation $X \rightarrow X$, if the number of cells XX_i is constant for the best and worst cases.

Combining Observations 1 and 2 yields an almost winged-edge-like C_3^9 data structure. As shown in Figure 5b, the storage and time trade-off has been linear thus far.

Baumgart discovered the use of a fractional relation in his design of the winged-edge data structure.[15] Instead of storing all the EF_i edges for a face F_i, he uses a face pointing to only one edge. The fractional relation $F \xrightarrow{1} E$ gives rise to a storage cost of F as it points to only one edge, and there are F of them. Similarly, $V \xrightarrow{1} E$ costs V in storage. The total storage is therefore $8E + (V+F) = 8E + (E+2) = 9E$. For time complexity, there are clearly three constant time queries. For the other six queries, local "clocking" around a face or around a vertex is required, costing EF_i or EV_i in time respectively. Since EV_i is of the same order as EF_i, we chose EV_i as the unit.[22] Hence, the total time complexity for the winged-edge is $6EV_i + 3k$. (See Figure 5c.) Note the dramatic improvement in time complexity from a global E to a local EV_i for a 25 percent increase in storage.

Symmetric data structure and its optimality

Suppose we add a new relation $V \rightarrow E$ to the C_2^9 data structure as shown in Figure 6b. The additional storage cost is $2E$. Let us examine the reduction in time. Because of the stored relations, there are clearly three constant-time queries. Next, notice the side benefit of an indirect relation $V \rightarrow F$, which can be derived from $V \rightarrow E$ and $E \rightarrow F$ to answer $T3$. From a given V_i, EV_i edges are returned in constant time. For each of the EV_i edges, exactly two faces are returned by $E \rightarrow F$ in constant time. Hence, the overall time for answering $T3$ is EV_i.

Consider a second side benefit of adding $V \rightarrow E$. Query $T1$ can be answered indirectly by following $V \rightarrow E$ and $E \rightarrow V$. From a given V_i, EV_i edges are retrieved. For each of the EV_i edges, exactly two vertices are retrieved, resulting in an overall time of EV_i for indirectly answering $T1$.

We achieve a third side benefit of adding $V \rightarrow E$ in that the addition permits an indirect relation of $E \rightarrow E$ via $E \rightarrow V$ and $V \rightarrow E$. For a given E_i, $E \rightarrow V$ returns exactly two vertices. For each of the two vertices, $V \rightarrow E$ returns all the EV_i edges in constant time. It takes EV_i time to identify the given E_i and pick out the two winged edges (one in the clockwise and the other in the counterclockwise direction). Therefore, the overall time to answer indirectly $E \rightarrow E$ is EV_i.

Observation 3
Add a relation $X \rightarrow Y$, if a relation $Y \rightarrow Z$, but not $X \rightarrow Z$, already exists.

Table 3.
Comparison of data structures.

Schema	Storage	Time
Winged edge	9E	$6EV_i + 3k$
Symmetric	8E	$5EV_i + 4k$

Applying Observation 3 once leads to a C_3^9 schema as shown in Figure 6b. Because of the side benefit of indirect query-answering, the additional storage cost of $2E$ (from $V \rightarrow E$) gives rise to not only an additional constant time query but three other EV_i time queries as well. Applying Observation 3 a second time leads to a C_4^9 schema as shown in Figure 6c. Its storage cost is $8E$, which is less than that of the winged edge. Its time complexity is $5EV_i + 4k$, which is also less than that of the winged edge.

It is useful to show, for a fixed storage cost of $8E$, that we have no other C_4^9 data structure that is faster than the one shown in Figure 6c—hence, its optimality. We give a comparison of all C_4^9 schemata in the Appendix. The complete enumeration of C_4^9 leads to the following theorem that asserts the optimality of the symmetric data structure shown in Figure 6c.

Theorem 3
The symmetric data structure is optimal in the C_4^9 class.

Conclusions

The results in the previous two sections suggest the comparison of data structures shown in Table 3. It is clear from Figure 7 that the symmetric data structure schema is faster yet less costly in storage than the winged edge.

To discover an even faster data structure, we can pursue two approaches: a special-purpose 3-D data structure designed for a specific set of applications or a general-purpose 3-D data structure that is globally optimal.

If the distribution of topological queries $T1$-$T9$ is available from a set of application programs, the relative frequencies of $T1$-$T9$ dictate the relations to be stored. For example, given a distribution of $T4$ (23%), $T6$ (21%), $T8$ (17%), $T7$ (12%), etc., one can design a data structure using $E \rightarrow V$, $E \rightarrow F$, $F \rightarrow E$, and $F \rightarrow V$. The result is that the query runs in constant time 73 percent of the time. The expected time for all queries can also be computed from their distribution.[22]

To construct a general-purpose 3-D data structure that is faster than the ones discussed here requires the following analysis. From the winged-edge schema, we see that a relation can be fractional; that is, we can store just one edge per face rather than all EF_i edges per face. A fractional relation creates a combinatorial explosion in the possible number of schemata. The number of possible designs is no longer C_m^9 but at least C_m^{18}. (For any relation $X \rightarrow Y$, we can store either all YX_i or just one of the neighbors.) In addition to the combinatorial complexity, we have analytic complexity. A globally optimal data structure must be as near the origin of the L-curve as possible. While it has been shown that the lower bound

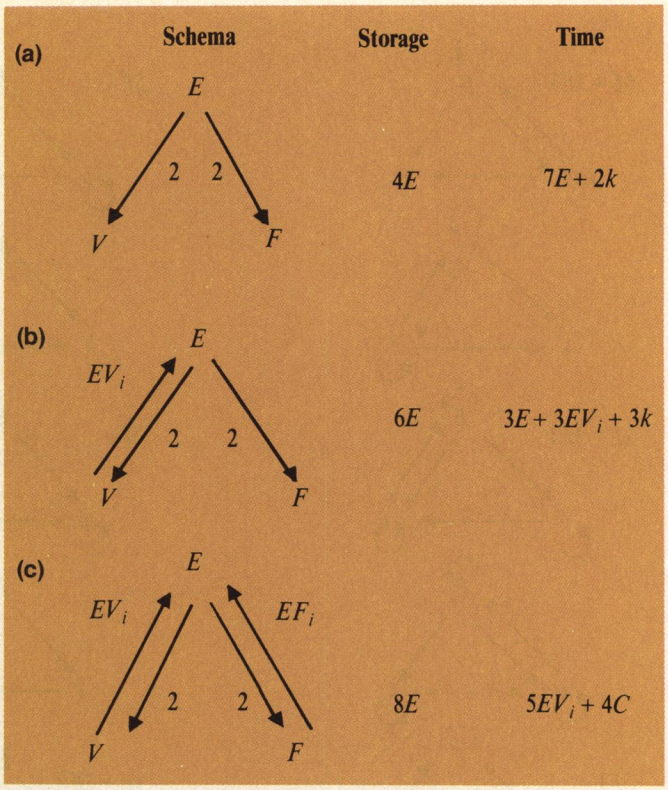


Figure 6. Development of symmetric data structure: C_2^9 (a); C_3^9 (b); and C_4^9 (c).

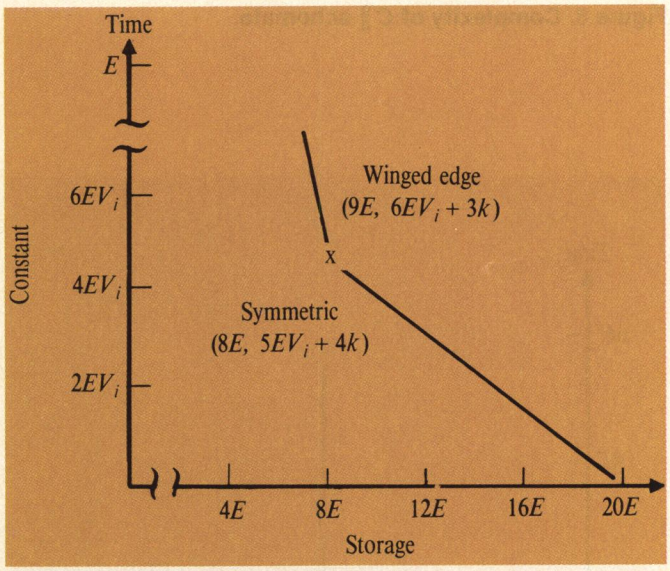


Figure 7. Lower portion of L-curve.

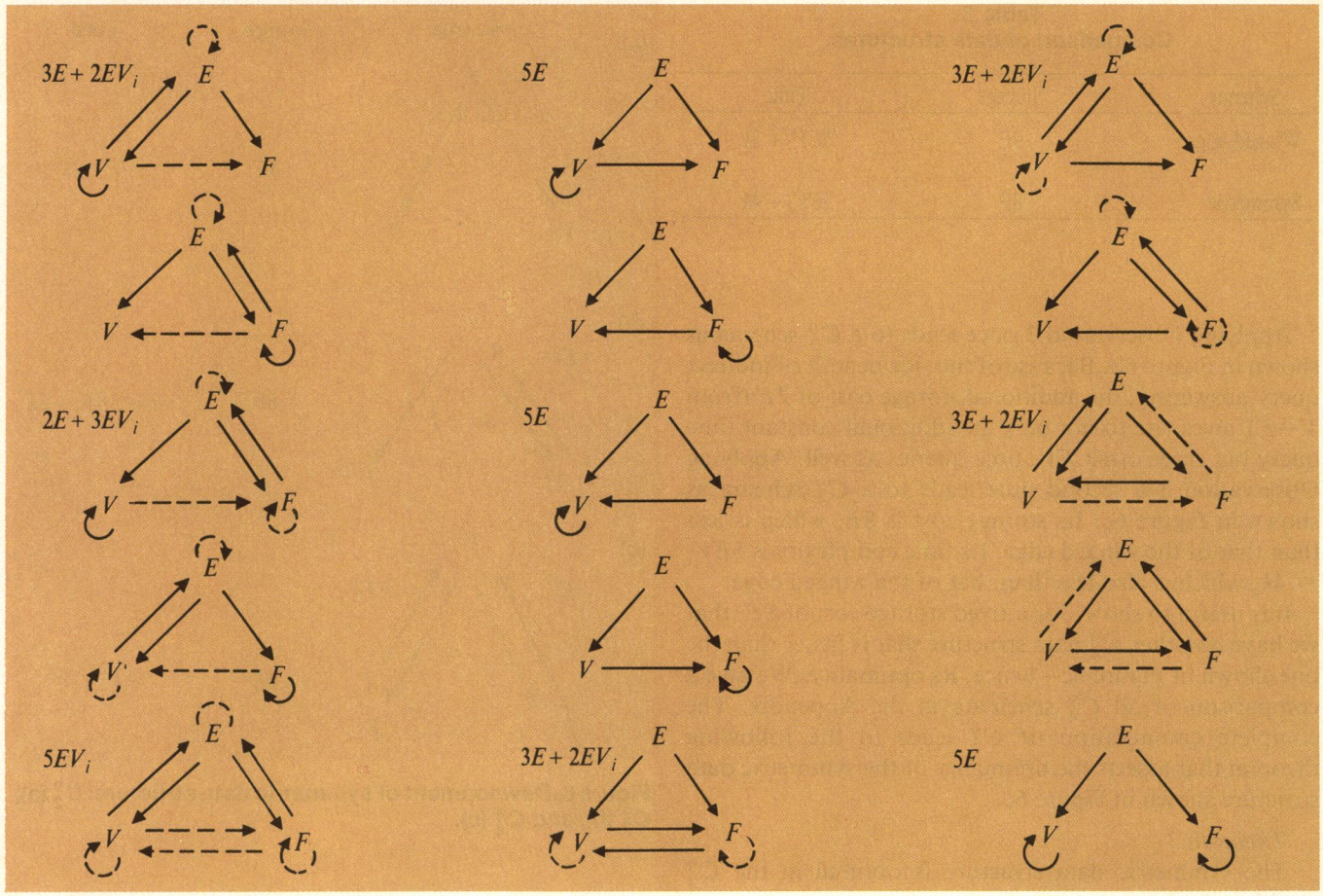


Figure 8. Complexity of C_4^9 schemata.

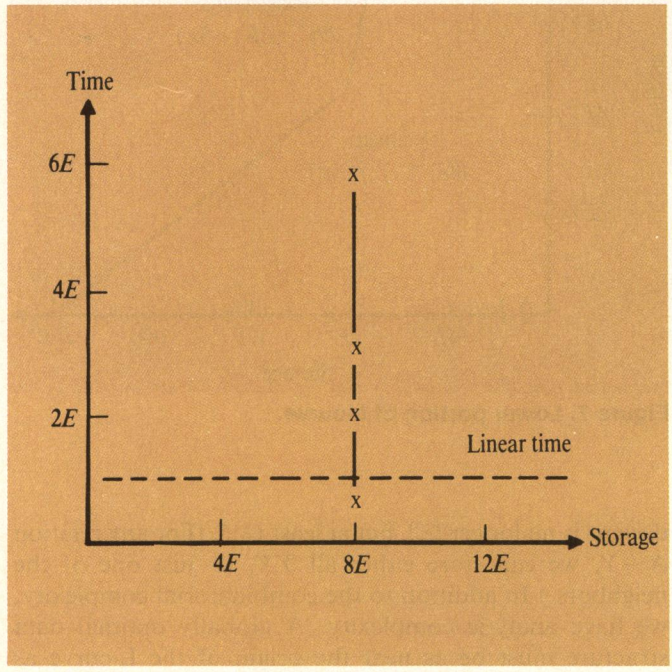


Figure 9. Storage and time of C_4^9 schemata.

for storage is $4E$, no nontrivial lower bound for time other than $9k$ for C_9^9 has yet been established. This remains a challenge. ■

Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research under contract F4920-82-C0089 and in part by IBM, Data Systems Division.

I wish to thank J. Wolter of the University of Michigan for his helpful suggestions, the four reviewers for their constructive comments, and T. Kunii, University of Tokyo, for encouragement.

Appendix: Analysis of C_4^9 schemata

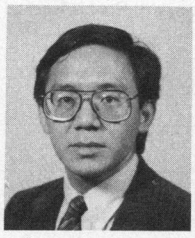
In the C_4^9 class, 126 combinations exist. Since four relations are stored, each costing $2E$ except $E \rightarrow E$, the minimum total storage cost should be $8E$.

Because $E \rightarrow E$ costs $4E$, we eliminate it, and C_4^8 combinations remain. By Observation 1, we see $E \rightarrow V$ and $E \rightarrow F$ are storage-cost effective. Now C_2^6, or 15, combinations of relations remain. Figure 8 shows the 15 schemata with the following organization. The first row permutes arcs leaving V. The second row is the dual of the first. The third row permutes arcs leaving V and F. The fourth row is the dual of the third. The fifth row is symmetric with respect to itself. The dashed arrows in the schemata indicate the indirect relations that can be derived in less than linear time. The time complexities are given below each schema.

Observe that the fastest C_4^9 schema is the first one in the fifth row, which is identical to the one shown in Figure 6c. The slowest, on the other hand, runs in linear time $5E$. The plot in Figure 9 shows that not all data structures having the same storage cost run equally fast.

References

1. I. C. Braid, "The Synthesis of Solids Bounded by Many Faces," *Comm. ACM,* Vol. 18, No. 4, Apr. 1975, pp. 209-216.
2. H. B. Voelcker and A. A. G. Requicha, "Geometric Modelling of Mechanical Parts and Processes," *Computer,* Vol. 10, No. 12, Dec. 1977, pp. 48-57.
3. B. Wordenweber, "Automatic Mesh Generation of 2 and 3 Dimension Curvilinear Manifolds," University of Cambridge Computer Laboratory, tech. report 18, Nov. 1981.
4. 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.
5. R. B. Tilove, "Extending Solid Modeling Systems for Mechanical Design and Kinematic Simulation," *IEEE Computer Graphics and Applications,* Vol. 3, No. 3, May 1983, pp. 9-19.
6. S. D. Roth, "Ray Casting for Modelling Solids," *Computer Graphics and Image Processing,* Vol. 18, 1982, pp. 109-144.
7. 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 Co., 1977, pp. 137-152.
8. 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.
9. T. Lozano-Perez, "Spatial Planning: A Configuration Space Approach," *IEEE Trans. Computers,* Vol. C-32, No. 2, Feb. 1983, pp. 108-120.
10. 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 J. Research and Development,* Vol. 24, No. 1, Jan. 1980, pp. 64-74.
11. A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods and Systems," *ACM Computing Surveys,* Vol. 12, No. 4, Dec. 1980, pp. 437-464.
12. D. Meagher, "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing,* Vol. 19, 1982, pp. 129-147.
13. A. A. G. Requicha and H. B. Voelcker, "Constructive Solid Geometry," University of Rochester, Production Automation Project, tech. memo 25, Nov. 1977.
14. I. C. Braid, "Six Systems for Shape Design and Representation," University of Cambridge, CAD Group document 87, May 1975.
15. B. G. Baumgart, "Winged-Edge Polyhedron Representation," Stanford University, Computer Science Department, report CS-320, Oct. 1972.
16. J. L. Bentley, "A Case Study in Applied Algorithm Design," *Computer,* Vol. 17, No. 2, Feb. 1984, pp. 75-88.
17. M. Gardner, "Aha! Gotcha: Paradoxes to Puzzle and Delight," W. H. Freeman and Co., San Francisco, 1982.
18. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading Mass., 1974.
19. 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.
20. A. Baer, C. Eastman, and M. Henrion, "Geometric Modelling: A Survey," *Computer-Aided Design,* Vol. II, No. 5, Sept. 1979, pp. 253-272.
21. I. C. Braid, "On Storing and Changing Shape Information," *Computer Graphics,* Vol. 12, No. 3, Aug. 1978, pp. 252-256.
22. T. C. Woo and J. D. Wolter, "A Constant Expected Time, Linear Storage Data Structure for Representing 3D Objects," *IEEE Trans. Systems, Man and Cybernetics,* Vol. SMC-14, No. 3, May/June 1984, pp. 510-115.



Tony C. Woo is associate professor in the Department of Industrial and Operations Engineering at the University of Michigan, Ann Arbor. From 1975 to 1977 he taught at the University of Illinois with a joint appointment in the Electrical, Mechanical and Coordinated Science Laboratory. His research interest is in computational geometry as applied to computer graphics, computer-aided manufacturing, and robotics.

Woo received his BS, MS, and PhD degrees in electrical engineering from the University of Illinois at Urbana. He is on the editorial board of the *International Journal of Computer Graphics* and is currently editing a special issue on computational geometry.

The author can be contacted at the University of Michigan, Dept. of Industrial and Operations Engineering, IOE Bldg., 1205 Beal Ave., Ann Arbor, MI 48109-2117.