Mesh Data Structures for Advanced Finite Element Applications

Mark W. Beall and Mark S. Shephard
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, NY 12180

1 Summary

Data structures to store information about the discretization of a geometric domain for the purpose of performing automated adaptive finite element analysis are defined by considering the information requirements of the processes that use them. A representation using a topological hierarchy consisting of vertices, edges, faces and regions, and containing classification information that relates the mesh back to the geometric model is described. A major strength of this representation is its generality and ease of application to varying procedures ranging from mesh generation to adaptive analysis processes. Three implementations of this representation are given which concentrate on different aspects of performance (storage requirements and speed). Comparisons are made to other published representations.

Key Words: mesh data structures

2 Introduction

A number of investigators are developing automated, adaptive finite element analysis procedures. A critical capability needed by these procedures is the ability to manipulate or query information about a discretization of the analysis domain, generally called a mesh.

Much has been written about data structures for certain parts of the adaptive finite element analysis process, such as various searching data structures for mesh generation [1,2,3], data structures for refinement of existing meshes [4,5,6,7,8,9], and data structures and algorithms for various parts of the solution process [10,11,12]. Much less has been written about data structures to store the mesh itself that are generally applicable to all parts of the process. Recently there have been some papers that presented alternative data structures [13,14,15] but that was not the main topic of the papers and thus there was not a complete analysis of the data structure.

There are weaknesses in the classic element-node approach [16,17] to mesh data structures, especially in an adaptive analysis environment. One major problem is the lack of information relating the mesh back to the original geometric model. This information, called classification [18,19], is critical for mesh generation and enrichment procedures [19,20,21], it allows the specification of analysis attributes in terms of the original geometric model rather than the mesh [22,23], which is important in adaptive analysis environments, and supports direct links to the geometric shape information of the original domain which is useful in p-version element integration procedures [24].

Many analysis and mesh modification procedures are easily written when a topological hierarchy of mesh entities is used to represent a mesh. For example, an edge based refinement procedure is much easier to write if it is possible to loop through all the edges in the mesh and get connectivity information about each of the edges.

An important goal in the development of a mesh data structure is ensuring its ability to effectively provide the information required by the various procedures that create and/or

use that data. The differing needs of such tools dictates that such a database must be very general and be able to answer all queries about the mesh. Such a general capability can only be achieved by utilizing a more general abstraction of a mesh, not the specific node-element view that was developed considering only the needs of performing a single analysis on a fixed mesh.

A number of alternative mesh data structures have been proposed for various forms of mesh adaptation [13,14,15]. Instead of immediately describing and comparing these structures, the requirements for a general purpose mesh database for automated adaptive finite element analysis on domains defined by manifold geometric models are given. The requirements are defined in terms of the type of information that must be obtained from such a database. A mesh database based on a topological hierarchy is then given. The efficiency of three implementations of this database is discussed with respect to both storage space and access time. A subsequent paper will describe extensions to the representation presented here for meshes of non-manifold geometric domains.

2.1 Nomenclature

The following notation is used throughout this document.

Models

Ω_V — Domain associated with the model V, $V = G, M$ where G signifies the geometric model and M signifies the mesh model.

$\overline{\Omega_V}$ — The closure of the domain associated with the model V, $V = G, M$.

Topological Entities

V_i^d — the ith entity of dimension d in model V. Shorthand for $V\{V^d\}_i$.

$\partial(V_i^d)$ — the entities on the boundary of V_i^d

$\overline{V_i^d}$ — closure of topological entity defined as $V_i^d \cup \partial(V_i^d)$.

$\sqsubset$ — classification symbol used to indicate the association of one or more entities from one model, typically M, with an entity in a higher model, typically G.

Groups

$\{V^d\}$ — unordered group of topological entities of dimension d in model V.

$\lfloor V^d \rfloor$ — ordered group of topological entities of dimension d in model V.

$[V^d]$ — cyclically ordered group of topological entities of dimension d in model V.

$\langle V^d \rangle$ — a group where the ordering is unspecified (ordering is one of: unordered, ordered or cyclically ordered).

φ_i — i^{th} topological entity in group φ, where φ is any one of the groups above.

Adjacency Operations

$\varphi\langle V^d \rangle$ — The set of entities of dimension d in model v that are adjacent to, or contained in φ. φ may be a single entity, V_i^d or $\langle V^d \rangle_i$, a group of entities, $\langle V^d \rangle$ (possibly a group resulting from another adjacency operation), or a model V.

$\varphi\langle V_{\pm}^d \rangle$ — An adjacency relation with directional use information associated with each entity. The $\pm$ indicates the directional use of each topological entity. A + indicates use in the same direction as the entity definition, a - indicates use in the opposite direction.

for example:

$V\{V^d\}$ — All of the entities of order d in model V

$V_i^{d_i}\{V^{d_j}\}$ The unordered group of topological entities of dimension d_j that are adjacent to (either on the closure of the entity $(j < i)$ or which it is on the closure of $(j > i)$) the entity ${}_vT_i^{d_i}$ in model V.

$V_i^{d_i}\{V^{d_j}\}_i$ The ith member of the unordered group of topological entities of dimension d_j that are adjacent to the entity ${}_vT_i^{d_i}$ in model V.

The adjacency notation is evaluated from left to right, for example:

$V_i^3\{V^0\}\{V^3\}_j$ is found by first finding the group $\varphi = T_i^3\{V^0\}$ and then the jth member of the group $\varphi\{V^3\}$.

3 Geometry Based Automated Adaptive Analysis

The goal of an analysis process is to solve a set of partial differential equations over the geometric domain of interest, $\overline{\Omega}_g$. Generalized numerical analysis procedures utilize a discretized version of this domain, called a mesh. Since the mesh domain, $\overline{\Omega}_m$, may not be identical to the original geometric domain, $\overline{\Omega}_g$, and/or various procedures, such as automatic mesh generation, adaptive mesh refinement and element stiffness integration, may need to understand the relationship of the mesh to the geometric model, it is critical to employ a representational scheme which can maintain the relationships between these two models. Although a number of schemes are possible for defining a geometric domain [25], the most advantageous for the current purposes are boundary-based schemes in which the geometric domain to be analyzed is represented as a set of topological types and adjacencies where each topological entity may have shape information associated with it. In the context of a domain representation, adjacencies are the relationships among topological entities which bound each other. For example, the edges that bound a face is a commonly used topological adjacency.

In addition to being unique, the use of topological entities and their adjacencies provides a convenient abstraction for defining the relationship of different models of the same domain. Boundary representations also allow the convenient specification, with respect to the geometric domain, of analysis attributes such as material properties, loads, boundary conditions and initial conditions [22,23]. An additional advantage of boundary representations is the fact that current computer aided design systems support a boundary representation of the domains defined within them. This allows the effective combination of these packages with automatic mesh generation. A final advantage of recent boundary representations are their ability to properly represent the non-manifold geometric domains commonly used for analysis processes [26,27].

3.1 Manifold and Non-Manifold Geometric Models

A manifold (two-manifold) model is one in which every point on a surface is homeomorphic to a two-dimensional disk [26]. In other words, if a point on a surface is examined closely enough the area around that point appears to be topologically flat (although not necessarily geometrically flat). A non-manifold model is one which has topological features that are not two-manifold. A common non-manifold situation is when an edge has more than two faces adjacent to it. Figure 1 shows two simple examples of non-manifold models that occur frequently in engineering analysis.

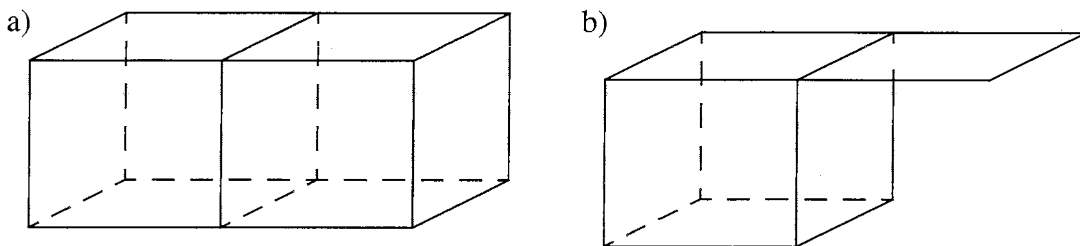


Figure 1. Examples of non-manifold models

One common situation where models like the one shown in Figure 1a will occur is when there are two different material regions that adjoin each other that are to be modeled. In this case there is an internal face that separates the two materials. The boundary of the internal face has non-manifold conditions. Situations like that shown in Figure 1b typically occur due to some idealization process. In this case the "hanging face" could have been some part with a small thickness that has been idealized to be a plate. Non-manifold models can arise from standard modeling operations on manifold geometries. The set of modeling operations known as Boolean set operations are not closed under manifold representations [28].

Over the past few years commercial geometric modelers have begun to correctly deal with non-manifold topology, so proper representation of these types of models will become more common in engineering analysis.

The following sections discuss the requirement for a general purpose mesh database for geometry based automated adaptive analysis. The present paper detail issues for meshes on manifold models and outlines some of the issues associated with non-manifold models. A subsequent paper will address the issues associated with meshes of non-manifold models in more detail.

4 Requirements for Meshes of Manifold Models

This section discusses the requirement for a mesh database to store meshes of manifold geometric models in an automated adaptive environment. The requirements given here are functional requirements that any implementation must provide. Particular implementations that meet these needs are given in the next section.

4.1 Topological Entities

Since the mesh is constructed from a geometric model that is represented using a boundary based scheme, using a similar scheme to also represent the mesh is advantageous for several reasons. First, topology provides an unambiguous, shape independent, abstraction of the mesh. Second, maintaining the relation between the model and the mesh is simplified since the same type of entities occur in both. Also having a hierarchy of topological entities representing the mesh allows many operations to be performed in more natural ways. For example, edge based refinement procedures are much more natural to write when it is possible to iterate through all the edges in the mesh.

Each topological entity of dimension d, M_i^d, is defined by a set of topological entities of dimension $d-1$, $M_i^d\{M^{d-1}\}$, which form its boundary. A region is a 3-d entity defined by the set of faces that bound it. A face is a 2-d entity defined by the set of edges that bound it. An edge is a 1-d entity defined by the two vertices that bound it. A vertex is a 0-d entity that is the base of the hierarchy, it has no lower order entities bounding it.

The proper consideration of general geometric domains also requires consideration of the loop and shell topological entities, and, in the case of non-manifold models, use entities for the vertices, edges, loops, and faces [26,28]. However, there are restrictions on the topology of a mesh that allows a reduced representation which is in terms of only the basic 0 to d dimensional topological entities. For the three-dimensional case (d=3) these entities are:

$$T_M = \{M\{M^0\}, M\{M^1\}, M\{M^2\}, M\{M^3\}\}$$

where $M\{M^d\}$, $d = 0, 1, 2, 3$ are respectively the set of vertices, edges, faces and regions defining the primary topological elements of the mesh domain.

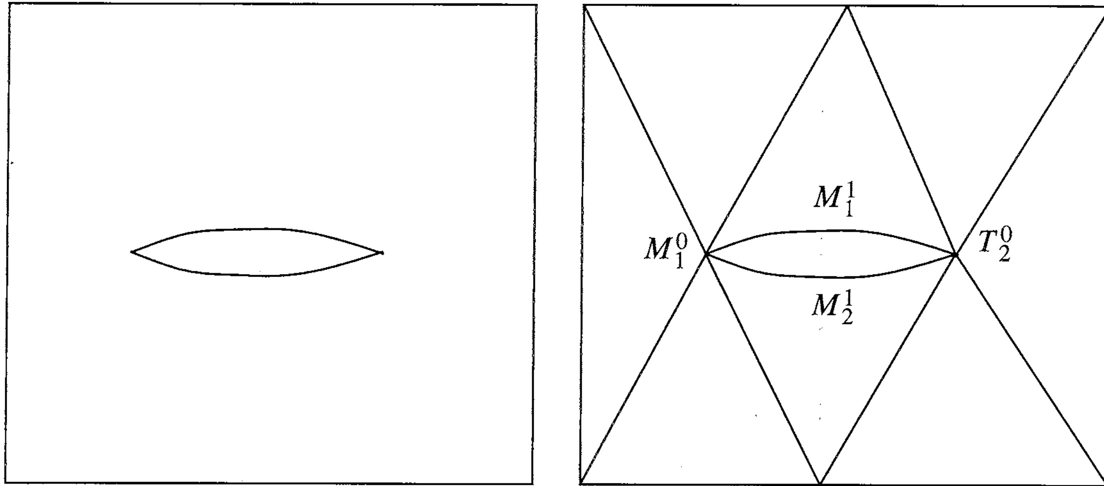
Restrictions on the topology of a mesh which allow this reduction include:

1. Regions and faces have no interior holes.
2. Each entity of order d_i in a mesh, M^{d_i}, may use a particular entity of lower order, M^{d_j}, $d_j < d_i$, at most once.
3. For any entity $M_i^{d_i}$ there is a unique set of entities of order $d_i - 1$, $M_i^{d_i}\langle M^{d_i-1}\rangle$ that are on the boundary of $M_i^{d_i}$ if at least one member of $M_i^{d_i}\langle M^{d_i-1}\rangle$ is classified on $G_j^{d_j}$ where $d_j > d_i$.

The first item means that regions may be directly represented by the faces that bound them, and faces may be represented by the edges that bound them. The shell and loop entities, required for general models, are therefore not needed in the mesh.

The second item allows an orientation of an entity to be defined in terms of its boundary entities (without the introduction of entity uses). For example, the orientation of an edge, M_i^1 which is bounded by vertices M_j^0 and M_k^0 may be uniquely defined as going from M_j^0 to M_k^0 only if $j \neq k$.

The third item means that an interior entity (defined as $M_i^{d_i} \sqsubset G_i^{d_j}$ where, $d_j \geq d_i$ and at least one of $\partial(M_i^{d_i}) \sqsubset G_i^{d_j}$) can be uniquely specified by the entities that bound it. This opens up the potential have an implementation that uses a reduced representation for the interior entities. This implementation is discussed later. This condition only applies to interior entities since it is possible for entities on the boundary of the model to have a non-unique set of boundary entities. A simple case of this is illustrated in Figure 2. Figure 2a shows a model of a plate with a hole. A coarse mesh of the plate (as might be used for a p-version analysis) is shown in Figure 2b.



(a) Geometric Model (b) Mesh

Figure 2. Example of mesh entities on the model boundary having non-unique boundary entities

The mesh is sufficiently coarse that the mesh topology and model topology are identical on the boundary of the opening. There are two edges M_1^1 and M_2^1 that form the mesh of the boundary of the opening. Both of these mesh edges have the same set of vertices, M_1^0 and M_2^0, thus the set of boundary entities for M_1^1 and M_2^1 is not unique.

Thus, a hierarchical representation consisting of mesh regions, M^3, mesh faces, M^2, mesh edges, M^1, and mesh vertices, M^0, is complete and sufficient to represent the topology of such a mesh. This hierarchical representation consists of each entity of higher order, M^i, being defined in terms of lower order entities, M^{i-1} which are on its boundary.

The existence of the topological entities in the mesh representation gives rise to these functional requirements for the mesh database:

Requirement 1. It must be possible to iterate through all the entities of a given type in a mesh.

Requirement 2. It must be possible to compare two entities to see if they are the same.

4.2 Classification

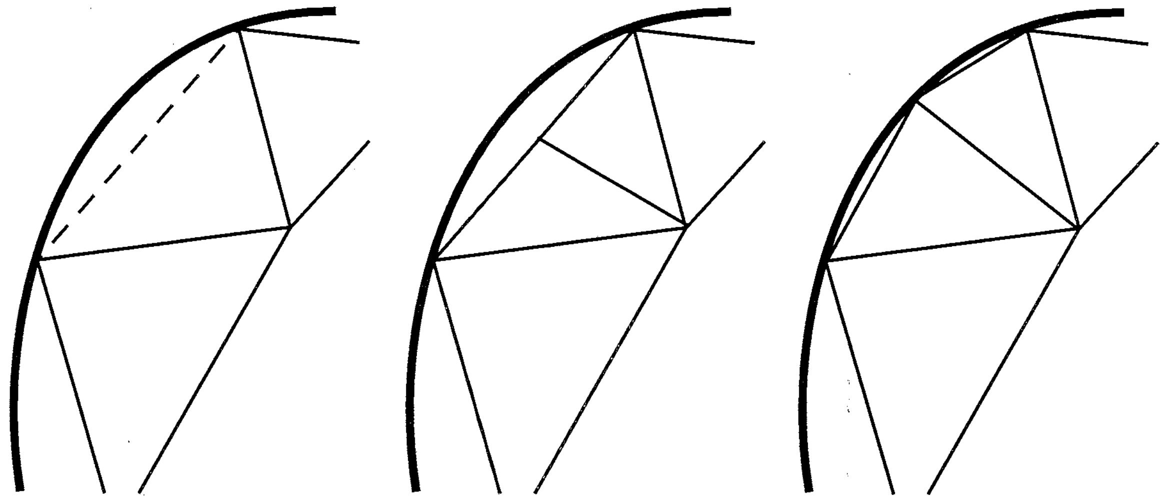
Critical to the understanding of the relationship of the mesh with the geometric domain is the concept of classification of a mesh with respect to its geometric model.

Definition: Mesh Classification Against the Geometric Domain - The unique association of a topological mesh entity of dimension d_i, $M_i^{d_i}$ *to a topological geometric domain entity of dimension* $d_j, G_j^{d_j}$ *where* $d_i \le d_j$, *is termed classification and is denoted* $M_i^{d_i} \sqsubset G_j^{d_j}$ *where the classification symbol,* $\sqsubset$ *, indicates that the left hand entity, or set, is classified on the right hand entity.*

Multiple $M_i^{d_i}$ can be classified on a $G_j^{d_j}$. A mesh region, M_i^3, is classified in the domain region, G_j^3, in which it lies. A mesh face, M_i^2, is classified in the domain region, G_j^3, or on the domain face, G_j^2, on which it lies. A mesh edge, M_i^1, is classified in the domain region, G_j^3 , on the domain face, G_j^2, or on the domain edge, G_j^1 , on which it lies. Finally, a mesh vertex, M_i^0, is classified in the domain region, G_j^3 , on the domain face, G_j^2, on the domain edge, G_j^1 , or on the domain vertex, G_j^0, on which it lies. Mesh entities are always classified with respect to the lowest order object entity possible.

Classification of the mesh against the geometric domain is central to (i) ensuring that the automatic mesh generator has created a valid mesh [18,19], (ii) transferring analysis attribute information to the mesh [22,23], (iii) supporting h-type mesh enrichments [21], and (iv) integrating to the exact geometry as needed by higher order elements [24]. An example of how classification information is used can easily be seen during the mesh refinement process illustrated in Figure 3.

In this example Figure 3a shows the mesh before the edge indicated by the dashed line is split to refine the mesh, the model edge is indicated by the bold line. Figure 3b shows the mesh after the edge has been split, splitting the adjacent face into two faces and making a



(a) before refinement (b) edge split (c) new vertex snapped to boundary

Figure 3. Edge split

new vertex on the edge. At this point the classification information can be used to recognize that the new vertex that was created is classified on the model edge (since the vertex was created by splitting an edge that was classified on the model edge). Since this situation is detected the new vertex can be "snapped" to the boundary so that it is located on the model. If the classification information does not exist this snapping to the boundary cannot be accomplished and then refining the mesh will not improve the geometric approximation that the mesh makes of the model.

The need for classification information gives the following functional requirement:

Requirement 3. It must be possible to retrieve the classification of any mesh entity

4.3 Geometric Information

In the geometric model each topological entity may have shape information associated with it that defines the geometry of the model. For the mesh, the geometric information that is required is limited to pointwise information in terms of the parametric coordinates

of the model entity that a mesh entity is classified on. Any other shape information that is needed to define the mesh can be obtained from the geometric model using the classification information and appropriate queries to the modeler (although for efficiency reasons, a particular implementation may choose to store some of this information rather than query it from the modeler). Zero or more locations may be stored for each mesh entity other than a mesh vertex which, by definition, must have exactly one spatial location associated with it. Thus, the following functional requirement:

Requirement 4. It must be possible to store spatial locations in terms of parametric coordinates on each mesh entity.

4.4 Adjacencies

Adjacencies describe how topological entities connect to each other. Certain adjacencies are also used to define higher order entities in terms of lower order ones (e.g. a face is defined by the edges that bound it).

There are natural orderings for several of the adjacencies which prove useful when accessing and manipulating the mesh. The forms of adjacencies used here are: an unordered list of entities adjacent to entity ϑ signified by $\vartheta\{\phi\}$, a linear list of entities adjacent to entity ϑ signified by $\vartheta\lfloor\phi\rfloor$, and a cyclic list of entities adjacent to entity ϑ signified by $\vartheta[\phi]$. For certain types of entities there is also a directional component to the adjacency relation that indicates how that entity is used in the specific adjacency. In these cases the right subscript, $\pm$, on the entity, $V^d_{\pm i}$, indicates a directional use of the topological entity as defined by its ordered definition in terms of lower order entities. A + indicates use in the same direction, while a - indicates use in the opposite direction (e.g. a face, M^2_i, could be defined by the set of edges bounding it as $M^2_i[M^1_{+i}, M^1_{-k}, M^1_{-l}]$ meaning that the bound-

ary of the face consists of edge M_j^1 used in the positive direction, from its first to second vertex, edge M_k^1 used in the negative direction and edge M_l^1 used in the negative direction).

Given the hierarchy of entities defined above there are many possible adjacencies that can be constructed. Although it is certainly not necessary to store all possible adjacencies, it is important to be able to efficiently retrieve any of them as they are needed. Section 6 of this paper discusses the trade-offs involved in storing various sets of adjacencies.

4.4.1 First-order adjacency relations

The most important set of relations are those which describe, for a given entity $M_k^{d_i}$, all of the entities, M^{d_j}, $(i \neq j)$ which are either on the closure of the entity $(j < i)$ or which it is on the closure of $(j > i)$. These are referred to as first order adjacencies. For example, the adjacency $M_i^2[M^0]$ is the circular ordered list of all of the mesh vertices which are on the closure of the mesh face M_i^2. The complete list of these adjacencies is as follows:

Vertex adjacencies: $M_i^0\{M^1\}$, $M_i^0\{M^2\}$, $M_i^0\{M^3\}$

Edge adjacencies: $M_i^1\lfloor M^0 \rfloor$, $M_i^1\{M^2\}$, $M_i^1\{M^3\}$

Face adjacencies: $M_i^2[M^0]$, $M_i^2[M_\pm^1]$, $M_i^2\lfloor M^3 \rfloor$

Region adjacencies: $M_i^3\{M^0\}$, $M_i^3\{M^1\}$,$M_i^3\{M_\pm^2\}$

Note that ordered, lower order adjacencies are used to define the orientation of higher order entities. The positive orientation of a mesh edge, M_i^1, is defined by the adjacency relation, $M_i^1\lfloor M^0 \rfloor$, the positive direction of the edge is going from the first vertex,

$M_i^1\lfloor M^0 \rfloor_0$, to the second vertex, $M_i^1\lfloor M^0 \rfloor_1$. This orientation of the edge is then used as part of the adjacency $M_i^2[M_{\pm}^1]$, as indicated by the $\pm$ symbol. Each face, M_i^2, uses its bounding edges in either the direction they are defined in or the opposite direction. Note that it is only necessary to actually store the directional part of the adjacency if a face has only two bounding edges, otherwise it can be found from the circular list of edges (a mesh face cannot have only a single bounding edge since that edge would then have to have the same vertex on each end, which is not permitted).

4.4.2 Second-order adjacency relations

Another set of important adjacency relations are those which describe, for a given entity $M_k^{d_i}$, all of the entities, M^{d_j}, which share any bounding entity of a given order, d_b with the entity. These are referred to as second-order adjacencies. An example of this is the adjacency, $M_i^3\{M^0\}\{M^3\}$, which is the set of all regions which share a vertex with M_i^3 (such a relationship is useful for element renumbering). The complete set of unordered second-order adjacencies can be expressed as follows:

$$M_k^{d_i}\{M^{d_b}\}\{M^{d_j}\}, d_j \neq d_b, d_i \neq d_b$$

As the notation suggests, the second order adjacencies can be derived from the first order adjacencies. Higher order adjacency relations can also be expressed in a similar manner.

To be able to retrieve any adjacency relations the following requirement must be satisfied:

Requirement 5. First order adjacencies must be retrievable for any mesh entity

4.5 Other Requirements

In addition to the requirements outlined thus far, the effective use of the mesh database for various types of needed operations necessitate the following requirements:

Requirement 6. It must be possible to uniquely associate arbitrary data with each entity.

By being able to associate data with each entity in the mesh, many algorithms can be written more efficiently. For example, an algorithm that traverses the mesh using the mesh adjacencies (e.g. starting at a face, get all the faces that are adjacent to that face through an edge and visit those adding their adjacent faces to the list to be visited, but visiting each face only once) can be made much more efficient by being able to mark entities that have been visited. Solution processes can use this to store their solution data directly on the mesh entities.

Requirement 7. Boundary edges and faces must be orientable.

Certain mesh generation operations can be made more efficient by ensuring that boundary edges and faces are oriented in the same direction as the model entity that they are classified on.

5 Examples of Using the Hierarchic Mesh Representation

Regardless of the particulars of the implementation, the representation described here can be used to satisfy any topological query about mesh information and about the relation of the mesh to the geometric model. By recasting some familiar queries to be in terms of mesh topological entities instead of in terms of elements and nodes we can see how these operators are used in an analysis environment. Having the topological entities and adjacencies available can also allow operations like the calculation of stiffness matrices to be done in a more general manner.

5.1 Finite Element Analysis - Stiffness Matrix and Face Load Calculation.

Suppose, for instance, that a region in a mesh, M_i^3, represents an element. In order to form the stiffness matrix for a linear element the following operations must be performed. First the vertices of the region, $M_i^3\{M^0\}$, are found. The location of each vertex can then be found by a simple query. Next the material properties to be used in the formulation are found. This is done by querying the classification of the mesh region and finding the model region that it is classified on. The material properties defined by attributes on the model regions can then be retrieved. Given this information the element stiffness matrix can be constructed in the usual manner. Note that specifying attributes on the model is a more natural way to specify attributes that are functions of position (e.g. layups for composite materials may be specified in a local coordinate system of the model, say, a cylindrical coordinate system for a laminated cylinder).

Since attributes such as loads are defined in terms of the geometric model the calculation of the equivalent nodal loads for an analysis can be done as follows. For each face in the geometric model check if there is a load associated with that face. If there is a load associated with the face then find all the mesh faces that are classified on that model face. For each mesh face calculate the equivalent nodal loads using the load distribution defined on the model face by evaluating the load distribution at the appropriate points on the mesh face.

5.2 Mesh Refinement - Face Split

One way to do mesh refinement is to modify an existing mesh by splitting and collapsing various regions. One such operator is called a "face split" where a new vertex is intro-

duced on a mesh face and each region using that face is split into three new regions. Here we consider splitting just the region on one side of the face. The extension to split both regions is trivial. Figure 4 illustrates this operation. The edges of the original tetrahedron are shown in bold and the new edges to be introduced are shown in normal weight. The new vertex to be introduced is labeled M^0_{new} in Figure 4 and is located on the front face.

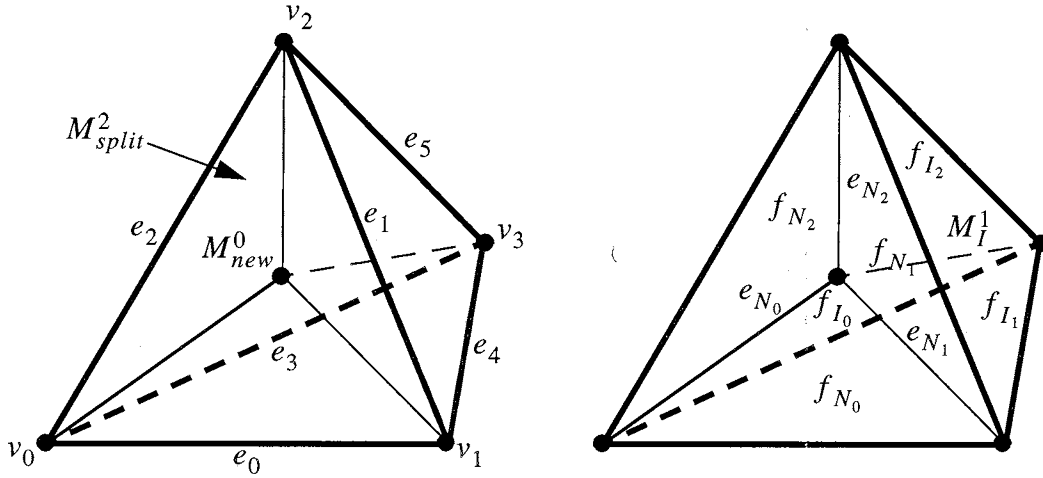


Figure 4. Face split operation

Given the face to split, M^2_{split}, and a vertex, M^0_{new}, on that face, snap the new vertex to the boundary if necessary:

if $M^2_{split} \sqsubset G^2_i$, reposition M^0_{new}, to be on G^2_i.

Define the sets of existing entities v, e, f as follows, these are the vertices, edges and faces of the region to be split as indicated in Figure 4.

$$v = \lfloor M^2_{split}[M^0], M^0_j \rfloor \text{ where } M^0_j \in M^3_{split}\{M^0\} \text{and } M^0_j \notin M^2_{split}\{M^0\}$$

$$e = \lfloor M^2_i[M^1], e_4, e_5, e_6 \rfloor \text{, where } e_i(i > 3) = M^1_{i-3}\lfloor v_i, v_3 \rfloor$$

$$f = \lfloor M^2_i \rfloor, i = 0 \ldots 2 \text{, such that } M^2_i \in \overline{M^3_{split}} \text{ and } e_i \in M^2_i \text{ and } M^2_i \neq M^2_{split}$$

create the following sets of new entities:

New edges on the face being split: $e_N = \left\lfloor M^1_{N_i}[M^0_{new}, v_i] \right\rfloor, i = 0 \ldots 2$

New faces on the face being split: $f_N = \left\lfloor M^2_{N_i}[e_i, e_{N_{mod(i+1,3)}}, e_{N_i}] \right\rfloor, i = 0 \ldots 2$

Edge interior to the region being split: $M^1_I[v_3, M^0_{new}]$

Faces interior to the region being split: $f_I = \left\lfloor M^{I_i}_2[e_{N_i}, M^1_I, e_{i+3}] \right\rfloor, i = 0 \ldots 2$

New regions: $M^3_{N_i}\{f_i, f_{I_{mod(i+1,3)}}, f_{I_i}, f_{N_i}\}, i = 0 \ldots 2$

6 Implementation Options

The adjacencies of various order mesh topological entities, and their classification with respect to the higher order models, are used to support a great number of the operations required by an automated adaptive analysis. Therefore, it is important that those adjacencies that are needed can be quickly determined. Clearly, if the adjacencies of each order entity against all other entities were stored, all possible adjacency information would be readily available. This approach would be highly wasteful with respect to the amount of data storage required. On the other hand, storing only a small number of adjacencies could require extensive searches and sorts to determine other specific adjacencies. An examination of the specific adjacencies used by the various algorithmic operations provides guidance as to the minimum number of adjacencies needed to be stored. For example references [8,13,14,15] define adjacencies used in specific finite volume and finite element procedures. Since the procedures considered here must support the highly demanding, from the view point of topological adjacencies, automatic mesh generation procedures,

and any form of adaptive analysis process on conforming unstructured meshes[1], all adjacencies are either stored, or can be quickly determined through a set of local traversals and sorts which are not a function of the mesh size.

To compare different implementations a common basis for comparison must be determined. Two issues of importance are storage space required and the time to access various adjacencies. Which of these is most important depends on the application. Four different schemes are defined below that may be used to obtain information from a given set of relations. They are defined in terms of how much computational work must be done to obtain the information. In the order of increasing expense, they are:

Retrieval - extracting a relation that is explicitly stored.

Collection - obtaining a relation that is not explicitly stored, but requires only collecting together information that is stored. For example, if we have $M_i^3\{M^2\}$ and $M_i^2\langle M^1\rangle$ stored, then finding $M_i^3\{M^1\}$ requires only collecting the M_i^1 information for each M_i^2 in $M_i^3\{M^2\}$ (in other words, finding $M_i^3\{M^2\}\{M^1\}$).

Local Searching - obtaining a relation that is not explicitly stored, but for which collection returns a superset of the entities satisfying that relation (i.e. some of the entities obtained by collection are not part of the relation). This requires that each entity be examined to determine whether it is to be included. For example, if we have $M_i^3\{M_{\pm}^2\}$, $M_i^2[M_{\pm}^1]$, $M_i^1[M^0]$ and $M_i^0\{M^3\}$ and want to find $M_i^2[M^3]$ for some face, the regions that result

1. A conforming mesh is one where all mesh entities exactly match. For example, a situation where the mesh edge bounding one mesh face has two mesh edges from other mesh faces lying exactly on top of it is not allowed. Although possible to extend the procedures presented here to support those situations, they are not considered in the present paper.

from collecting $M_i^2[M_{\pm}^1][M^0]\{M^3\}$ contains regions that are not bounding regions for the given face, thus it is necessary to check each region to see if the face is on its boundary.

Global Searching - searching through all the entities of a given type and checking each one to see if it satisfies the given relation.

The process of retrieval is $O(1)$, collection and local searching are both $O(n_e)$, where n_e is the number of entities that must be examined to find the relation. For both collection and local searching n_e is proportional to the size of the adjacency relation, $n_e = cn_a$, it is independent of the number of entities in the mesh. For collection, the typical range for c for the relations described in this paper is, $2 < c < 6$. For local searching the range is $4 < c < 25$. In addition, local searching requires a check on each entity to see if some condition is true, so in general it will take longer than collection. Global searching takes time proportional to the number of entities in the mesh which makes it an unacceptable way to find any relation. For example, if the relation $M_i^2\lfloor M^3 \rfloor$ could only be found by global searching, then to find the two regions bounding a face in a 1 million region mesh, would require traversing the list of every mesh region and checking each face on that region to see if it was the correct one.

Given a graph of the first order adjacencies (Figure 5), it can be easily seen what property of the stored adjacency relations is necessary to avoid global searching. From any node in the graph it must be possible to traverse the graph in some manner such that any other node can be reached. In other words there must be one or more cycles in the graph that include all four of the nodes.

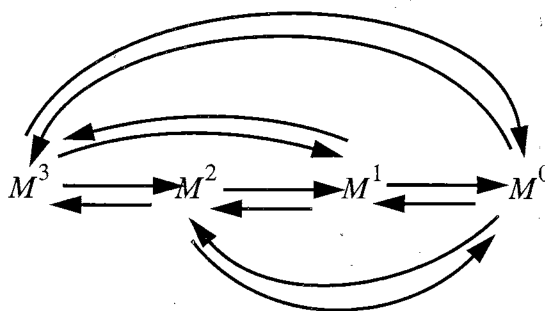


Figure 5. Graph of first order adjacencies

6.1 Storage Requirements

The data structure described here supports a mesh that may be any mixture of various shaped entities (tets, hexes, wedges, pyramids, triangles, quads, lines, etc.). For the purpose of examining and comparing storage requirements it is simplest to consider meshes that have only one type of top-level entity (a top-level entity is one that has no higher order entities adjacent to it).

The amount of storage required for the topological relations of a mesh varies depending on the type of discretization and the adjacencies stored. Two example cases are considered here: an all tetrahedral mesh and an all hexahedral mesh. It is expected that a mesh that is a mixture of tetrahedrons, hexahedrons and other common elements would have storage requirements between these two. The number of pointers that are needed to store each type of connectivity for the two types of discretization are shown in Table 1. This table shows the number of members in the relation $M_k^{d_i}\langle M^{d_j}\rangle$, $(i \neq j)$ for the entire mesh in terms of the number of entities in the mesh. $N_M^d \equiv |M\{M^{d_i}\}|$, is the number of entities of dimension d in model m. For example, in a tetrahedral mesh, there are a total of $4N_M^3$ pointers from regions to faces since each region points to four faces. This means that there are also

$4N_M^3$ pointers from faces to regions since each face that is pointed to by a region points back to that region (although each face only points to two regions).

Table 1. Adjacency storage requirements

Tetrahedral Mesh

	M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$4N_M^3$	$6N_M^3$	$4N_M^3$
M_i^2	$4N_M^3$		$3N_M^2$	$3N_M^2$
M_i^1	$6N_M^3$	$3N_M^2$		$2N_M^1$
M_i^0	$4N_M^3$	$3N_M^2$	$2N_M^1$	

Hexahedral Mesh

	M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$6N_M^3$	$12N_M^3$	$8N_M^3$
M_i^2	$6N_M^3$		$4N_M^2$	$4N_M^2$
M_i^1	$12N_M^3$	$4N_M^2$		$2N_M^1$
M_i^0	$8N_M^3$	$4N_M^2$	$2N_M^1$	

For each of these types of meshes the relationship between the number of the various entities in the mesh is shown in Table 2.

Table 2. Relations between number of entities in mesh

Tetrahedral Mesh	$N_M^2 \approx 2N_M^3$, $N_M^1 \approx \frac{6}{5}N_M^3$, $N_M^0 \approx \frac{4}{23}N_M^3$
Hexahedral Mesh	$N_M^2 \approx 3N_M^3$, $N_M^1 \approx 3N_M^3$, $N_M^0 \approx N_M^3$

The relationships in Table 2 are approximate since they are derived assuming an infinite mesh plus the assumptions that follow. For a tetrahedral mesh there are $4N_M^3$ pointers from faces to regions, since each face points to exactly two regions (in an infinite mesh) the relation $2N_M^2 = 4N_M^3$ must hold, so $N_M^2 = 2N_M^3$. The rest of the relations are derived assuming an infinite mesh of equilateral tetrahedrons. An equilateral tetrahedron has 70.529 degree dihedral angles, thus any edge completely surrounded by tetrahedra has approximately 5 tetrahedra attached to it (note, however, that it is actually impossible to

have a mesh comprised of all equilateral tetrahedrons since they do not close pack). Using this and the fact that there are $6N_M^3$ pointers from edges to regions gives $5N_M^1 = 6N_M^3$ or $N_M^1 = \frac{6}{5}N_M^3$. Similarly, the solid angle formed at the vertex of an equilateral tetrahedron has a solid angle of 0.5514 steradians. Thus there are approximately $4\pi/0.5514 = 22.78 \approx 23$ tetrahedrons that meet at a vertex. Thus $23N_M^0 = 4N_M^3$ or $N_M^0 = \frac{4}{23}N_M^3$. These values were checked by calculating ratios of the number of entities in real meshes and the following ranges were found: $2.02 < N_M^2/N_M^3 < 2.19$, $1.2 < N_M^1/N_M^3 < 1.45$, $0.18 < N_M^0/N_M^3 < 0.27$, which shows reasonably good agreement with the numbers in Table 2. For a hexahedral mesh an infinite regular mesh was assumed. In this case each face is adjacent to two regions, each edge is adjacent to four faces and four regions and each vertex is adjacent to six edges, eight faces and eight regions.

Using the relations in Table 2 (and their inverses), the adjacency storage requirements, Table 1, is rewritten in terms of the number of regions in the mesh (Table 3) and the number of vertices in the mesh (Table 4).

Table 3. Connectivity storage requirements in terms of regions

Tetrahedral Mesh

	M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$4N_M^3$	$6N_M^3$	$4N_M^3$
M_i^2	$4N_M^3$		$6N_M^3$	$6N_M^3$
M_i^1	$6N_M^3$	$6N_M^3$		$2N_M^3$
M_i^0	$4N_M^3$	$6N_M^3$	$2N_M^3$	

Hexahedral Mesh

	M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$6N_M^3$	$12N_M^3$	$8N_M^3$
M_i^2	$6N_M^3$		$12N_M^3$	$12N_M^3$
M_i^1	$12N_M^3$	$12N_M^3$		$6N_M^3$
M_i^0	$8N_M^3$	$12N_M^3$	$6N_M^3$	

Table 4. Connectivity storage requirements in terms of vertices

Tetrahedral Mesh

	M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$23N_M^0$	$35N_M^0$	$23N_M^0$
M_i^2	$23N_M^0$		$35N_M^0$	$35N_M^0$
M_i^1	$35N_M^0$	$35N_M^0$		$14N_M^0$
M_i^0	$23N_M^0$	$35N_M^0$	$14N_M^0$	

Hexahedral Mesh

	M_i^3	M_i^2	M_i^1	M_i^0
M_i^3		$6N_M^0$	$12N_M^0$	$8N_M^0$
M_i^2	$6N_M^0$		$12N_M^0$	$12N_M^0$
M_i^1	$12N_M^0$	$12N_M^0$		$6N_M^0$
M_i^0	$8N_M^0$	$12N_M^0$	$6N_M^0$	

Another piece of information that can be extracted from the above information is the average number of adjacencies of each entity type to each other type on a per entity basis. This is given in Table 5.

Table 5. Average number of adjacencies per entity $|M_i^{row}\{M^{col}\}|$

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		4	6	4
M_i^2	2		3	3
M_i^1	5	5		2
M_i^0	23	35	14	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		6	12	8
M_i^2	2		4	4
M_i^1	4	4		2
M_i^0	8	12	6	

There are many subsets of the first order adjacencies that could be stored and from which the remaining first order (and thus any other) adjacencies can be derived. The next three sections give implementations that match well with the various requirements for retrieving information used in automated adaptive analysis processes.

6.2 One Level Adjacency Representation

One set of relationships that can effectively meet these requirements is to maintain adjacencies between entities one dimension apart. A data structure similar to this is discussed in Reference 29 for the specific case of tetrahedral meshes. Figure 6 graphically depicts this set of relationships.

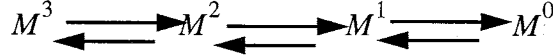


Figure 6. Graph of adjacencies for one-level adjacency representation

The actual adjacencies stored are:

Downward Adjacencies: $M_i^1\lfloor M^0 \rfloor$, $M_i^2[M_\pm^1]$, $M_i^3\{M_\pm^2\}$

Upward Adjacencies: $M_i^0\{M^1\}$, $M_i^1\{M^2\}$, $M_i^2\lfloor M^3 \rfloor$

The missing relations can be reconstructed as follows:

$$M_i^3\{M^1\} = M_i^3\{M^2\}\{M^1\}$$

$$M_i^3\{M^0\} = M_i^3\{M^2\}\{M^1\}\{M^0\}$$

$$M_i^0\{M^3\} = M_i^0\{M^1\}\{M^2\}\{M^3\}$$

$$M_i^0\{M^2\} = M_i^0\{M^1\}\{M^2\}$$

$$M_i^1\{M^3\} = M_i^1\{M^2\}\{M^3\}$$

$$M_i^2[M^0] = \{M_i^2[M_{\alpha_n}^1]_n \lfloor M^0 \rfloor_{\beta_n}\}, \beta_n = \begin{cases} 0, \alpha_n = + \\ 1, \alpha_n = - \end{cases}, n = 0 \ldots |M_i^2[M^1]|$$

The complexity of this last expression deserves a bit of explanation. The adjacency $M_i^2[M^0]$ is found by considering each edge in the adjacency $M_i^2[M_\pm^1]$ in sequence. One

vertex from each edge will be added to the set depending on the direction the edge is being used by the face. If edge $M_i^2[M_{\pm}^1]_n$ is being used in the + direction then the first vertex of the edge is taken, if it is being used in the - direction then the second vertex of the edge is taken. This results in the ordered set of vertices around the face.

This reduced adjacency set can give us all the adjacency information that is required without global searches. Note that this adjacency set only requires collection to obtain the missing relations.

The time to retrieve a relation by collection is less than that implied by the operators used above to describe the process. For example, obtaining $M_i^3\{M^0\} = M_i^3\{M^2\}\{M^1\}\{M^0\}$ for a tetrahedron requires looking at only two of the faces of the region to determine all of the vertices. The first face yields three of the vertices of the region and any other face gives the fourth. Similar processes can be determined for some of the other relations that need to be found for any of the shapes of interest. Table 6 shows an estimate of the operation count needed to obtain each of the adjacency relations for the one-level representation.

Table 6. Operation count for retrieving adjacency $M_i^{row}\{M^{col}\}$ for one-level representation

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1	9	6
M_i^2	1		1	3
M_i^1	10	1		1
M_i^0	140	70	1	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1	20	16
M_i^2	1		1	4
M_i^1	8	1		1
M_i^0	48	24	1	

6.3 Circular Adjacency Representation

Another set of adjacencies that meet the requirement of being able to provide all of the adjacency information, without global searching, is to store downward pointers from each entity to the entity one dimension lower and to store pointers from the vertices up to the highest order entities that are using them (in a 3-D manifold mesh this would be the mesh regions) as shown in Figure 7.

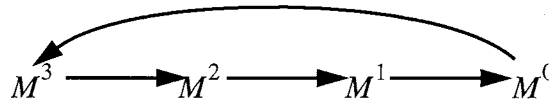


Figure 7. Graph of adjacencies for circular adjacency representation

The actual adjacencies stored are:

Downward Adjacencies: $M_i^1[M^0]$, $M_i^2\langle M_\pm^1 \rangle$, $M_i^3\{M_\pm^2\}$

Upward Adjacencies: $M_i^0\{M^3\}$

The obvious advantage of this scheme over the previous one is that less information is stored. The disadvantage is that more work must be done to obtain the upward adjacencies that are not being stored.

This set of relations provides the minimum connectivity storage in which all entities are explicitly represented. This can be seen by weighting the corresponding edges in the graph of first order adjacencies (Figure 5) with the connectivity storage requirements in Table 3. This set of relations (or the similar one using the inverse of each relation: $M_i^0\{M^1\}$, $M_i^1\{M^2\}$, $M_i^2\lfloor M^3 \rfloor$ and $M_i^3\{M^0\}$) is the minimum weighted cyclic path that includes all four nodes in the graph (which is the requirement to avoid global searching). The rea-

son that the three downward and one upward adjacencies are used instead of three upward and one downward is that there is directional use information in the downward relations.

The task of finding the missing relations is more involved than with the up/down adjacency relations. The procedures for constructing the adjacency relations $M_i^0\{M^1\}$, $M_i^1\{M^2\}$, and $M_i^2\lfloor M^3 \rfloor$ are shown below. The remaining adjacency relations are found in the same manner as shown for the one-level representation previously.

$M_i^0\{M^1\}$:

$M_j^1 \in M_i^0\{M^1\}$ if $M_j^0 \in \partial(E_k)$ where $E = M_i^0\{M^3\}\{M^2\}\{M^1\}$

$M_i^1\{M^2\}$:

$M_j^2 \in M_i^1\{M^2\}$ if $M_j^1 \in \partial(F_k)$ where $F = M_i^1\{M^0\}\{M^3\}\{M^2\}$

$M_i^2\lfloor M^3 \rfloor$:

$R = M_i^2\{M^1\}\{M^0\}\{M^3\}$ (R is the set of all regions bounding the closure of M_i^2)

Each region in R must be checked to determine if it is adjacent to M_i^2. The direction that the region is using M_i^2 determines which side of M_i^2 the region is on:

$M_i^2\lfloor M^3 \rfloor_0 = R_j$ such that $R_j\{M^2_{\beta_k}\}^k = M_i^2$ and $\beta_k = -$

$M_i^2\lfloor M^3 \rfloor_1 = R_j$ such that $R_j\{M^2_{\beta_k}\}^k = M_i^2$ and $\beta_k = +$

Table 7 shows an estimate of the operation count needed to retrieve each adjacency. The downward adjacencies can all be retrieved using either retrieval or collection, most of the upward adjacencies require local searching consisting of traversing the entire cycle in the

adjacency graph then doing topological queries on each entity that is found. Thus, the time required to determine these relations is substantially longer than that required to find the same relations in the one-level adjacency. However, determination of any relationship is still independent of the number of entities in the mesh.

Table 7. Operation count for retrieving adjacency $M_i^{row}\{M^{col}\}$ for circular representation. Method needed to retrieve adjacency indicated as: (r) = retrieval, (c)= collection, (ls) = local searching

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1 (r)	9 (c)	6 (c)
M_i^2	299 (ls)		1 (r)	3 (c)
M_i^1	538 (ls)	570 (ls)		1 (r)
M_i^0	1 (r)	264 (ls)	304 (ls)	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		1 (r)	20 (c)	16 (c)
M_i^2	148 (ls)		1 (r)	4 (c)
M_i^1	304 (ls)	296 (ls)		1 (r)
M_i^0	1 (r)	192 (ls)	228 (ls)	

6.4 Reduced Representations

The third restriction on the topology of a mesh given earlier was: For any entity $M_i^{d_i}$ there is a unique set of entities of order $d_i - 1$, $M_i^{d_i}\langle M^{d_{i-1}}\rangle$ that are on the boundary of $M_i^{d_i}$ if at least one member of $M_i^{d_i}\langle M^{d_{i-1}}\rangle$ is classified on $G_i^{d_j}$ where $d_j > d_i$. One implication of this restriction is that interior entities can be uniquely defined by their boundary entities. This gives the ability to eliminate some interior entities without losing any information about the mesh, specifically we can eliminate interior faces and edges. Also, by relaxing the requirement to support meshes where multiple mesh entities, classified on the boundary of the model, are bounded by the same mesh entities (such as the situation shown in Figure 2), a situation which occurs in very coarse meshes, some entities on the boundary can be eliminated, at least with manifold models.

It is important to note that the functionality that was presented earlier for the mesh database must not be affected by the elimination of the entities, that is, although the implementation of the database does not explicitly represent these entities, the interface to the database must act as though it does. It still must be possible to iterate through all the entities in the mesh (including those that are no longer explicitly represented), to obtain any adjacency information that is needed, to compare two entities $M_i^{d_i}$ and $M_j^{d_i}$ to see if they are the same and to associate data with an entity even if it is not explicitly represented. The general idea is that if an entity that is not represented and the program using the database needs that entity (e.g. the entity is returned as a part of an adjacency relation) a temporary proxy is returned for the entity, the lifetime of this proxy is only as long as the program is referencing that entity.

There are two important issues in eliminating entities. First, it must be possible to reconstruct the eliminated entities on the fly, as needed. Second, it must be possible to associate data with the eliminated entities.

6.4.1 Reconstructing eliminated entities

The most complex aspect of the process of reconstructing eliminated entities is dealing with the fact that edges and faces are oriented entities which must always have a consistent orientation. That is, if an edge, M_i^1, which is not explicitly stored is defined as $M_i^1\lfloor M_j^0, M_k^0 \rfloor$ at one point it must never be redefined as $M_i^1\lfloor M_k^0, M_j^0 \rfloor$ on a later query. The same consistency of ordering of edges around a face applies.

One way in which this can be accomplished is as follows:

1. Number each vertex in the mesh uniquely.

2. An edge M_i^1 is defined as going from M_j^0 to M_k^0 where $j < k$. That is, edges which are not explicitly represented are defined such that the positive direction of the edge is from the lower numbered vertex to the higher numbered one.
3. A face M_i^2 is defined in terms of an ordered set of vertices $M_i^2[M^0]$ where the face orientation is defined by the loop in the direction from the lowest numbered vertex to the next lowest numbered vertex adjacent to the lowest vertex (note the quadrilateral face in Figure 8 is defined as 5-8-7-12 since the lowest numbered vertex in the face, adjacent to vertex 5, is 8).

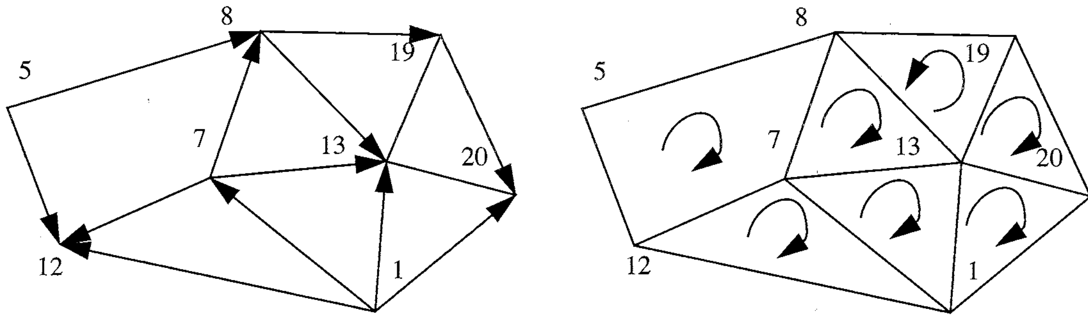


Figure 8. Edge and face orientations based on vertex numbering

So given a set of vertices that define a face, a unique orientation is defined, the edges and the direction the face is using the edges is also defined. Note that in doing this we have lost the ability to arbitrarily orient interior edges and faces. Since the orientation of an interior edge or face is determined by the numbering of the vertices (which is hidden from the programmer) an edge defined from vertex M_k^0 to vertex M_j^0, $M_i^1\lfloor M_k^0, M_j^0 \rfloor$, may actually end up being oriented as $M_i^1\lfloor M_j^0, M_k^0 \rfloor$ if $j < k$. This cannot be fixed by simply renumbering the vertices since it is impossible to arbitrarily orient edges as shown in Figure 9. However, since there is no requirement to orient internal edges and faces (since the orientation must satisfy no requirement other than being consistent) this is a workable approach. An alternative approach is to store a piece of data, as described below, associated with each face or edge that indicates if the actual direction of the entity is the same as or opposite to the direction its vertex ordering defines.

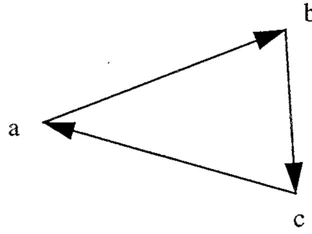


Figure 9. Impossible edge orientation, requires a<b, b<c, c<a

It is also necessary to know which vertices are used to define the edges and faces. This information comes from the region definition. In order to infer the existence of faces and edges the relation $M_i^3\lfloor M^0 \rfloor$ (the ordered set corresponding to $M_i^3\{M^0\}$) must be defined for each type of region (e.g. hexahedron, tetrahedron, wedge, etc.). This requirement was not present in the previous representations where the "shape" of the region did not need to be explicitly stored. From this ordered set of vertices and the known shape of the region, the set of vertices which form interior edges and faces can be determined. For example, Figure 10 shows two tetrahedral regions, one defined by $M_i^3\lfloor M^0 \rfloor = \lfloor M_4^0, M_7^0, M_3^0, M_5^0 \rfloor$ and the other by $M_j^3\lfloor M^0 \rfloor = \lfloor M_4^0, M_3^0, M_8^0, M_5^0 \rfloor$. From this it can be determined that there is a face defined by vertices M_4^0, M_7^0 and M_3^0 and an edge defined by vertex M_4^0 and vertex M_5^0 but not by vertex M_8^0 and vertex M_7^0.

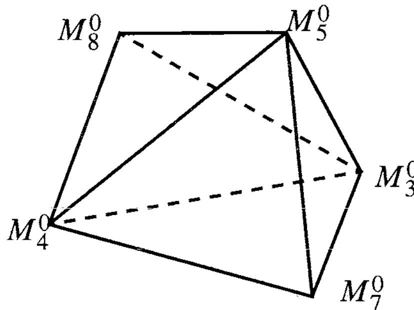


Figure 10. Edge and face determination from $M_i^3\lfloor M^0 \rfloor$ adjacency

6.4.2 Storing data on eliminated entities

The second issue in eliminating entities is that it is no longer possible to actually store data on them since they don't exist in a permanent form. The best way to resolve this seems to be to store the data associated with the edges and faces on the vertices used to define them. To do this not only the data must be stored but also information that indicates which face or edge the data belongs to (for an edge it would be necessary to store the other vertex, for a face all the other vertices which define the face). This extra information used to indicate the owner of the data is only needed when there is data actually stored on the entity.

6.4.3 Implementations

There are two possibilities for implementation of the reduced data structure. The first eliminates faces and edges only in the interior of the mesh, this will be called the reduced interior representation. On the boundary $M_i^{d_i} \sqsubset G_i^{d_i}$ is represented (that is faces classified on faces and edges classified on edges must be represented, but edges classified on faces need not be). This representation will meet all of the requirements given earlier. The adjacency graph of this representation is shown in Figure 11.

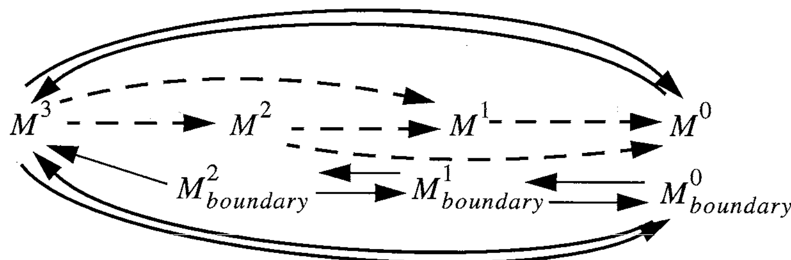


Figure 11. Adjacency graph for reduced interior representation

The adjacency graph is somewhat more complicated due to the fact that the mesh representation is now heterogeneous, different entities exist on the boundary than exist on the

interior. The dashed lines in the graph indicate adjacencies that are implicitly stored due to the ordering of vertices defining a region. The ordering of vertices which defines the faces must not be used for faces on the boundary. Local searching must be done to find these faces (which are represented). It can be seen from the adjacency graph that any downward adjacency can be directly retrieved. Upward adjacencies are obtained in a similar manner to the circular hierarchic representation. Table 8 shows an estimate of the operation count to retrieve adjacencies for the reduced representation. The counts shown only consider retrieving adjacencies for interior entities, more searching must be done on the boundary to find boundary entities which are represented. It is assumed that it takes one operation to construct a proxy to stand in for an entity that is not explicitly represented. The operation counts for the same operation are less than those for the circular representation since all of the downward adjacencies are stored (either explicitly or implicitly). The retrieval operations that require local searching still take more time than the same operation using the one-level adjacency representation.

Table 8. Operation count for retrieving adjacency $M_i^{row}\{M^{col}\}$ for reduced representation. Method needed to retrieve adjacency indicated as: (r) = retrieval, (c)= collection, (ls) = local searching

Tetrahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		4 (r)	6 (r)	1 (r)
M_i^2	293 (ls)		3 (r)	1 (r)
M_i^1	230 (ls)	373 (ls)		1 (r)
M_i^0	1 (r)	219 (ls)	198 (ls)	

Hexahedral Mesh

	M^3	M^2	M^1	M^0
M_i^3		8 (r)	12 (r)	1 (r)
M_i^2	176 (ls)		4 (r)	1 (r)
M_i^1	112 (ls)	212 (ls)		1 (r)
M_i^0	1 (r)	116 (ls)	86 (ls)	

A slight modification of the allowable topology of a mesh opens up another implementation option. Changing the third restriction on the topology of a mesh given in Section 4.1 to:

> For any entity $M_i^{d_i}$ there is a unique set of entities of order $d_i - 1$, $M_i^{d_i}\langle M^{d_i - 1}\rangle$ that are on the boundary of $M_i^{d_i}$.

adds the restriction that boundary entities must also have a unique set of lower order entities bounding them. This makes the mesh in Figure 2b (and its equivalent with two faces that share a common set of edges, which looks like a pillow) an invalid mesh, but allows the elimination of faces and edges everywhere in the mesh. The adjacency graph for this representation is given in Figure 12.

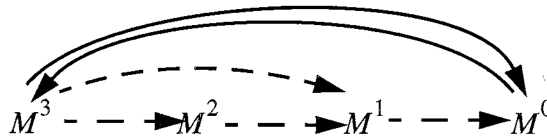


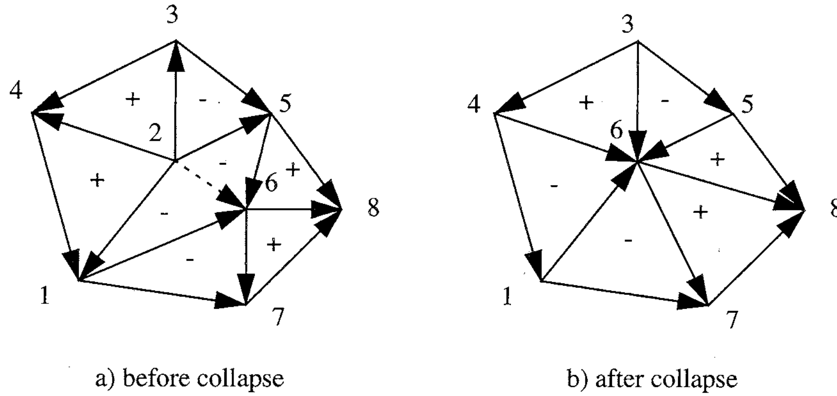
Figure 12. Adjacency graph for reduced representation with no boundary entities

Although this option seems to be simpler than the other reduced representation, there is a fair amount of hidden information that is being stored as data on the mesh rather than being stored as part of the definition of a mesh entity. The two major things being stored as data are the classification and orientation of the boundary entities. This results in this representation not actually saving as much space as it might appear to. In fact the total savings over the reduced interior representation amount to, at most, five percent. Due to this the reduced interior representation will be the only form considered further in this paper.

6.4.4 Issues with the reduced representations

There are some issues with the reduced representations that make them less desirable than the representations with a complete set of entities. The most important problem is that modifying the mesh may change the definition of mesh entities that were not directly modified. For example, Figure 13 shows an edge collapse procedure where this redefinition occurs. The dashed edge (between vertex 2 and 6) is being collapsed with vertex 6 replacing vertex 2. The two figures show the edge orientations (as arrows on the edges) and the face orientations (a + face has its normal pointing out of the page, a - face has it pointing into the page) as given by the vertex numbering scheme described earlier. After the edge collapse, three things have happened that would not happen if the edges and faces were directly represented: i) two edges (2-4 and 2-3) have actually changed their identities causing any references to them to become invalid, ii) those same two edges have changed their orientations (thus the direction that the faces are using them have changed), and iii) a face (previously 1-2-4, now 1-4-6) has changed its orientation. The same operation using a representation with all entities present would have resulted in only adjacency information being changed. The entities themselves would have remained unchanged including their orientations.

This non-local effect makes these representations much less efficient for doing many mesh modification procedures. This is due to the fact that information about the topology of the mesh that is saved by the procedure may become invalid when an operation on the mesh is performed. This means that the procedure must reacquire this information after each mesh modification. With the full representation of all entities, the propagation of these changes is very limited and predictable.



a) before collapse b) after collapse

Figure 13. Edge collapse

7 Comparison to Classic FE Data Structure

This section compares the size of a data structure based on the classic element-node connectivity to one based on the hierarchic representations discussed in this paper. The reason for doing this comparison is to show that the hierarchic data structure does not necessarily take significantly more storage space than a classic data structure, especially when other data structures needed to perform an analysis are considered. It should be noted that the comparison is not really a fair one since the classic data structure does not meet the needs of various adaptive procedures. However it is an obvious comparison that should be done to understand what price must be paid to meet these needs. Three different implementations of the hierarchic data structure are compared to the classic data structure. Meshes consisting of tetrahedral and hexahedral elements of various order (up to cubic) are considered. For the purposes of the comparison, only serendipity elements with nodes on edges will be considered, although all the data structures can easily be extended to any type of element.

7.1 Classic Mesh Data Structure

The classic approach to mesh data structures, here referred to as the "Classic Data Structure", describes the mesh in terms of elements and nodes. In addition, there are additional data structures that may be needed for operations such as node or element reordering which are discussed below. As shown in Figure 14, an element is defined by an ordered list of nodes. Each node has an id and a position in space.

```
Element {                              Node {
    int type;                              int id;
    ptr attributes;                        real x,y,z;
    ptr nodes[n]                       }
}
```

n = 4(linear tet.) 10 (quad. tet.), 16 (cubic tet.), 8 (linear hex), 20 (quad. hex.), 32 (cubic hex.)

Figure 14. Classic mesh data structure

The size of the Element data structure is $n + 2$ where n is the number of nodes in the element. The size of the Node data structure is 7 (sizes are given in words, where an integer or a pointer is one word and a real value is 2 words).

Table 9 shows the number of nodes (N) as a function of the number of edges and vertices in a mesh.

Table 9. Number of nodes and elements in mesh

Element Type	**Number of Nodes**	**Number of Elements**
Linear	${}_mN^0$	N_M^3
Quadratic	${}_mN^1 + {}_mN^0$	N_M^3
Cubic	$2{}_mN^1 + {}_mN^0$	N_M^3

Given the size and number of nodes and elements, the storage needed for the mesh can be calculated as shown in Table 10.

Table 10. Total storage by entity - classic

	Element	Node	Total
	Tetrahedral		
Linear	$6N_M^3$	$1N_M^3$	$7N_M^3$
Quadratic	$12N_M^3$	$8N_M^3$	$20N_M^3$
Cubic	$18N_M^3$	$15N_M^3$	$33N_M^3$
	Hexahedral		
Linear	$10N_M^3$	$7N_M^3$	$17N_M^3$
Quadratic	$22N_M^3$	$28N_M^3$	$50N_M^3$
Cubic	$34N_M^3$	$49N_M^3$	$83N_M^3$

However this is not all the storage needed to perform an analysis. Additional storage may be needed to perform other operations such as equation renumbering and storage of the global stiffness matrix.

7.1.1 Equation Renumbering

The actual amount of storage needed to perform equation renumbering depends on the algorithm used. Some of the most successful algorithms are based on graph theory, among these are Sloan, Gibbs-King, Gibbs-Poole-Stockmeyer (GPS) and reverse Cuthill-McKee (see reference 30 and references therein). All of these algorithms must build up a graph of the node-to-node connectivity of the finite element mesh. The storage for this graph depends on the implementation but an efficient implementation using an adjacency list accessed by a pointer vector [31] requires storage of $2E+N$ words, where E is the number of edges and N is the number of nodes in the graph. Other storage is also needed by the various reordering algorithms which varies greatly by algorithm. The number of nodes, N, in the adjacency graph is the same as the number of finite element nodes in the

mesh. The number of edges in the graph depends on the type of mesh and on the particular mesh itself.

The value of E can be calculated for various types of meshes. For each node the number of edges in the graph connecting to that node, E_n, is the number of nodes on all the elements that share that node (counting each node only once). For linear hexahedral elements (nodes only at the vertices) the connectivity of each node is all the nodes on the eight hexahedral elements that meet at each vertex, this can be easily found to be 26. The value of E for various types and orders of meshes is shown in Table 11.

Table 11. Node connectivity

	Tetrahedral		Hexahedral	
Element Order	**Vertex Nodes**	**Edge Nodes**	**Vertex Nodes**	**Edge Nodes**
Linear	14	N/A	26	N/A
Quadratic	61	22	80	50
Cubic	107	38	130	81

The total value of E is then the connectivity of each node times the number of nodes of that type. This value is shown in Table 12. The total storage for the classic data structure including renumbering is given in Table 13.

Table 12. Total connectivity storage

Element Type	Tetrahedral	Hexahedral
Linear	$2.5N_M^3$	$26N_M^3$
Quadratic	$37N_M^3$	$230N_M^3$
Cubic	$64N_M^3$	$371N_M^3$

Table 13. Total storage for classic data structure including renumbering

Element Type	Tetrahedral	Hexahedral
Linear	$9.5N_M^3$	$43N_M^3$
Quadratic	$57N_M^3$	$280N_M^3$
Cubic	$97N_M^3$	$454N_M^3$

Note the dramatic increase in connectivity information that must be stored for higher order elements. These numbers indicate that in these situations it may be wise to avoid these renumbering schemes which are derived solely from the structure of the assembled system of equations and use an approach that is based on the connectivity of the mesh as is described in Section 7.5.

7.2 Hierarchic Data Structure - One-Level

The hierarchic data structure for the one-level representation that is used to compare to the classic data structure is shown in Figure 15. This data structure includes the full hierarchy of mesh entities. Note that the number of upward pointers from edges to faces and from vertices to edges varies for each edge and vertex in the mesh. The number shown here is the average number of entities in that adjacency relation as was derived earlier

Based on this data structure the size of each entity can be calculated as shown in Table 14 (again assuming that pointers and integers are one word and real numbers are two words). Total storage for the mesh is shown broken down by mesh entity in Table 15.

Table 14. Entity sizes for one-level representation

	Tetrahedral					Hexahedral			
	Region	Face	Edge	Vertex		Region	Face	Edge	Vertex
Linear	6	7	9	23		8	8	8	15
Quadratic	6	7	16	23		8	8	15	15
Cubic	6	7	23	23		8	8	22	15

```
Region {
    ptr classification;
    int #faces;
    ptr faces[4^t or 6^h];
}

Face {
    ptr classification;
    int #edges;
    ptr edges[3^t or 4^h];
    ptr regions[2];
}

Edge {
    ptr classification;
    ptr vertices[2];
    int #faces;
    ptr faces[5^t or 4^h]
    int node_id[0^l,1^q or 2^c];
    Point node_location[0^l,1^q or 2^c];
}

Vertex {
    ptr classification;
    #edges;
    edges[14^t or 6^h]
    int node_id;
    Point location;
}

Point {
    real x,y,z;
}
```

Meaning of superscipts:
t: tetrahedral mesh
h: hexahedral mesh
l: linear mesh
q: quadratic mesh
c: cubic mesh

Figure 15. Hierarchic data structure - one-level

Table 15. Total storage by entity - one-level representation

	Region	**Face**	**Edge**	**Vertex**	**Total**
			Tetrahedral		
Linear	$6N_M^3$	$14N_M^3$	$11N_M^3$	$4N_M^3$	$35N_M^3$
Quadratic	$6N_M^3$	$14N_M^3$	$19N_M^3$	$4N_M^3$	$43N_M^3$
Cubic	$6N_M^3$	$14N_M^3$	$28N_M^3$	$4N_M^3$	$52N_M^3$
			Hexahedral		
Linear	$8N_M^3$	$24N_M^3$	$24N_M^3$	$15N_M^3$	$71N_M^3$
Quadratic	$8N_M^3$	$24N_M^3$	$45N_M^3$	$15N_M^3$	$92N_M^3$
Cubic	$8N_M^3$	$24N_M^3$	$66N_M^3$	$15N_M^3$	$113N_M^3$

7.3 Hierarchic Data Structure - Circular

The hierarchic data structure for the circular representation is shown in Figure 16. This data structure again includes the full hierarchy of mesh entities. The number of upward pointers from vertices to regions varies for each edge and vertex in the mesh. The number shown here is the average number of entities in that adjacency relation as was derived earlier. A real implementation would have to be a little more complicated to handle mesh

generation and adaption procedures since in a partially constructed mesh the vertices may need to point to entities other than regions.

Region {
  ptr classification;
  int #faces;
  ptr faces[4^t or 6^h];
}

Face {
  ptr classification;
  int #edges;
  ptr edges[3^t or 4^h];
}

Edge {
  ptr classification;
  ptr vertices[2];
  int #faces;
  int node_id[0^l,1^q or 2^c];
  Point node_location[0^l,1^q or 2^c];
}

Vertex {
  ptr classification;
  #regions;
  regions[23^t or 8^h]
  int node_id;
  Point location;
}

Point {
  real x,y,z;
}

Meaning of superscipts:
t: tetrahedral mesh
h: hexahedral mesh
l: linear mesh
q: quadratic mesh
c: cubic mesh

Figure 16. Hierarchic data structure - circular

The sizes of each entity are shown in Table 16. In comparison to the one-level representation the region is the same size, the face and edge structures are smaller (since they do not have upward connectivity stored) and the vertex is larger (since the number of regions adjacent to a vertex is larger than the number of edges adjacent to a vertex).

The overall storage broken down by entity is shown in Table 17. The total storage is 15-25% less than the one-level representation.

Table 16. Hierarchic representation entity sizes - circular

	Tetrahedral				**Hexahedral**			
	Region	Face	Edge	Vertex	Region	Face	Edge	Vertex
Linear	6	5	4	32	8	6	4	17
Quadratic	6	5	11	32	8	6	11	17
Cubic	6	5	18	32	8	6	18	17

Table 17. Total storage by entity - circular

	Region	Face	Edge	Vertex	Total
			Tetrahedral		
Linear	$6N_M^3$	$10N_M^3$	$5N_M^3$	$5N_M^3$	$26N_M^3$
Quadratic	$6N_M^3$	$10N_M^3$	$13N_M^3$	$5N_M^3$	$34N_M^3$
Cubic	$6N_M^3$	$10N_M^3$	$22N_M^3$	$5N_M^3$	$43N_M^3$
			Hexahedral		
Linear	$8N_M^3$	$18N_M^3$	$12N_M^3$	$17N_M^3$	$55N_M^3$
Quadratic	$8N_M^3$	$18N_M^3$	$33N_M^3$	$17N_M^3$	$76N_M^3$
Cubic	$8N_M^3$	$18N_M^3$	$48N_M^3$	$17N_M^3$	$91N_M^3$

7.4 Hierarchic Data Structure - Reduced Interior Representation

The data structure for the reduced interior representation (Figure 17) is more complicated than the other two hierarchic representations. In this implementation there are no interior edges or faces, however there are two different representations of the vertex, one for the boundary and one for the interior. The vertices stored in the region must be stored in a known order for each shape of element (e.g. tetrahedron, hexahedron). The data structure shown assumes that there is a full representation on the boundary (edges classified on model faces are represented). Again, for the purposes of obtaining sizes for the entities, where there is a variable number of adjacencies to be stored, the average number is shown. The implementation shown here is a little simpler than would be needed for mesh generation since it would be necessary to be able to represent a partially constructed mesh. This could be done by using mesh edges and faces on the interior when there are no higher order entities. These data structures needed to support the mesh generation functions are not shown here but would be similar to the edges and faces in the one-level representation

```
Region {
    ptr classification;
    int type;
    ptr vertices[4^t or 8^h];
}

Boundary Face {
    ptr classification;
    int #edges;
    ptr edges[3^t or 4^h];
    ptr regions[2];
}

Boundary Edge {
    ptr classification;
    ptr vertices[2];
    int #faces;
    ptr faces[2]
    int node_id[0^l,1^q or 2^c];
    Point node_loc[0^l,1^q or 2^c];
}

Boundary Vertex {
    ptr classification;
    # b_edges;
    ptr b_edges[6^t or 4^h]
    int node_id;
    Point location;
    int #regions;
    ptr regions[12^t or 4^h];
    int #interior edges;
    Edge_info edges[4^t or 1^h];
}

Vertex {
    ptr classification;
    #regions;
    regions[23^t or 8^h]
    int node_id;
    Point location;
    Edge_info edges[7^t or 3^h]
}

Edge_info{
    ptr other_vertex;
    int node_id[1^q or 2^c]
    Point[1^q or 2^c];
}

Point {
    real x,y,z;
}
```

Meaning of superscipts:
t: tetrahedral mesh
h: hexahedral mesh
l: linear mesh
q: quadratic mesh
c: cubic mesh

Figure 17. Hierarchic data structure - reduced interior

without the upward adjacency information. These additional structures are not necessary for procedures which refine or derefine an existing mesh.

The sizes of each entity are given in Table 18. Compared to the other two hierarchic representations most of the data has been moved to the vertex. Part of the reason for this is that

where information (nodes in this case) was stored on the edges it is now stored on one of the vertices of the edge.

Table 18. Entity sizes - reduced interior

	Region	Boundary Face	Boundary Edge	Boundary Vertex	Vertex
	Tetrahedral				
Linear	6	7	6	29	32
Quadratic	6	7	13	61	88
Cubic	6	7	20	89	137
	Hexahedral				
Linear	10	8	6	19	17
Quadratic	10	8	13	27	41
Cubic	10	8	20	34	62

Calculating the total storage for this representation is more difficult since the storage for entities is different if they are on the boundary or on the interior and the ratio of boundary to interior entities depends on the particular mesh. Looking at information on some large (>100,000 regions) tetrahedral meshes it was found that the percentage of boundary vertices ranged from 10% to 65%, typically being close to 30%, and the percentage of boundary edges ranged from 5% to 35%, typically being close to 10%, boundary faces were typically less than 5%. Thus, for the comparison used here it is assumed that the mesh has 30% of its vertices, 10% of its edges and 5% of its faces on the boundary. This leads to the storage shown in Table 19.

Table 19. Total storage by entity - reduced interior

	Region	Boundary Face	Boundary Edge	Boundary Vertex	Vertex	Total
			Tetrahedral			
Linear	$6N_M^3$	$0.5N_M^3$	$1N_M^3$	$1.5N_M^3$	$4N_M^3$	$13N_M^3$
Quadratic	$6N_M^3$	$0.5N_M^3$	$1.5N_M^3$	$3N_M^3$	$11N_M^3$	$22N_M^3$
Cubic	$6N_M^3$	$0.5N_M^3$	$2.5N_M^3$	$5N_M^3$	$17N_M^3$	$31N_M^3$
			Hexahedral			
Linear	$10N_M^3$	N_M^3	$2N_M^3$	$6N_M^3$	$12N_M^3$	$31N_M^3$
Quadratic	$10N_M^3$	N_M^3	$4N_M^3$	$8N_M^3$	$29N_M^3$	$52N_M^3$
Cubic	$10N_M^3$	N_M^3	$6N_M^3$	$10N_M^3$	$44N_M^3$	$71N_M^3$

7.5 Node Renumbering with the Hierarchic Mesh Representations

As mentioned earlier, extra data structures for node (or element) renumbering are not needed with the hierarchic mesh representation. The reason for this is that the node-to-node (or element-to-element) adjacency information that is built up in those procedures is already available from the hierarchic data structure.

One simple procedure for nodal renumbering based off the hierarchic mesh representation is shown below. This procedure uses the existing adjacency information to traverse the mesh, numbering the nodes as it does so. A small amount of storage is needed for the queue and possibly to find a good starting set of vertices, but extra storage for the node-to-node connectivity is not needed. Experience has shown that this type of renumbering results in global stiffness matrices with bandwidths competitive with those generated by other renumbering algorithms. In fact, the algorithm is much the same as reverse Cuthill-McKee [30], it is even possible to add degree of node (number of connections to other nodes) priority to this algorithm making it even more like reverse Cuthill-McKee.

```
initialize queue with vertices
current_node_number = number of nodes
while queue not empty{
     remove first vertex from queue
     number node at vertex with current_node_number
     current_node_number = current_node_number -1
     for each unnumbered node on any higher order entities adjacent to vertex {
             number node with current_node_number
             current_node_number = current_node_number -1
     }
     add neighboring vertices of vertex that are not in queue to queue
}
```

This type of renumbering could also be used with a classic data structure. It would require building the node-element connectivity for the vertex nodes only.

7.6 Size Comparison

The information from the previous sections is summarized in Table 20 and Table 21. As can be seen, in the majority of cases, there is a cost in storage space for most of the hierarchic data structures presented in this paper. The cost decreases rapidly as higher order elements are used. If renumbering is taken into account the hierarchic data structures are smaller for quadratic and higher order elements.

Table 20. Size comparison - tetrahedral meshes (numbers in parenthesis are classic data structure with renumbering information)

Element Order	**Classic**	**One-Level**	**% of Classic**	**Circular**	**% of Classic**	**Reduced Interior**	**% of Classic**
Linear	$7N_M^3$ ($9.5N_M^3$)	$35N_M^3$	500% (368%)	$26N_M^3$	371% (274%)	$13N_M^3$	186% (137%)
Quadratic	$20N_M^3$ ($57N_M^3$)	$43N_M^3$	215% (75%)	$34N_M^3$	170% (60%)	$22N_M^3$	110% (39%)
Cubic	$33N_M^3$ ($97N_M^3$)	$52N_M^3$	158% (54%)	$43N_M^3$	130% (44%)	$31N_M^3$	94% (32%)

Table 21. Size comparison - hexahedral meshes (numbers in parenthesis are classic data structure with renumbering information

Element Order	Classic	One-Level	% of Classic	Circular	% of Classic	Reduced Interior	% of Classic
Linear	$17N_M^3$ ($43N_M^3$)	$71N_M^3$	418% (165%)	$55N_M^3$	324% (128%)	$31N_M^3$	182% (72%)
Quadratic	$50N_M^3$ ($280N_M^3$)	$92N_M^3$	184% (33%)	$76N_M^3$	152% (27%)	$52N_M^3$	104% (19%)
Cubic	$83N_M^3$ ($454N_M^3$)	$113N_M^3$	136% (25%)	$91N_M^3$	110% (20%)	$71N_M^3$	86% (16%)

Another interesting result can be found by normalizing the mesh sizes by the number of nodes in the mesh as shown in Table 22. Since the number of nodes in the mesh is related to the amount of information stored on the mesh during the solution process, this can be viewed as the information cost of the mesh. Doing this normalization allows meshes of different element orders and element types to be compared. As would be expected, increasing the element order decreases the information cost since the fixed cost of storing the mesh topology is amortized over more nodes. One interesting observation is the high information cost of a linear tetrahedral mesh compared to a linear hexahedral mesh and that the large difference virtually disappears when the order of each mesh is raised to quadratic.

Table 22. Information cost (words/node) (numbers in parenthesis are classic data structure with renumbering information)

Element Order	Classic	One-Level	Circular	Reduced Interior
Tetrahedral Mesh				
Linear	40 (56)	201	153	76
Quadratic	15 (41)	31	25	16
Cubic	13 (37)	20	17	12
Hexahedral Mesh				
Linear	17 (43)	71	55	31
Quadratic	13 (70)	23	19	13
Cubic	12 (65)	16	13	10

Another item that must be considered is that, during a solution process, other information must be stored in addition to the mesh. At a minimum, a certain number of degrees of freedom per node are stored. At most all the local stiffness matrices may be stored. Somewhere in the middle, in terms of storage, would be storing the assembled stiffness matrix in some form. This storage is relevant since, if it is large compared to the mesh storage required, the extra storage needed to use a representation of the mesh that requires more storage may not be very significant.

A summary of this storage is given in Table 23. Storage is given for three different purposes. First, the storage for the degrees of freedom that hold the solution itself is fixed at 2 words per degree of freedom (one double precision number). Second, storage needed for the individual element matrices is given, as might be used by an iterative solver with an

element level preconditioner where all the element matrices are stored. Third, the storage needed for an assembled global stiffness matrix using compressed row storage [32] is given assuming a symmetric system (only half of the global matrix is actually stored). The compressed row storage is likely to be the most compact storage possible for the global matrix, since it only stores the nonzero terms and has little overhead. Using a skyline or other storage method would require more storage. In particular, a skyline storage requires storage per node equal to the average bandwidth of the matrix which will increase as the mesh is refined for a given problem, the compressed row storage per node is independent of the problem size. The last two items in Table 23 depend on the square of the number of degrees of freedom per node, since the size of the stiffness matrix of an element scales in this manner.

Table 23. Information cost for solution data structures (words/node) n is the number of degrees of freedom per node

Element Order	**Solution**	**Element Matrices**	**Global Stiffness**
Tetrahedral Mesh			
Linear	$2n$	$376n^2$	$21n^2$
Quadratic	$2n$	$292n^2$	$41n^2$
Cubic	$2n$	$398n^2$	$64n^2$
Hexahedral Mesh			
Linear	$2n$	$256n^2$	$39n^2$
Quadratic	$2n$	$400n^2$	$86n^2$
Cubic	$2n$	$585n^2$	$132n^2$

Note that while the information cost for the mesh decreases as the element order is increased the information cost for the solution generally increases.

To see how this compares to the mesh storage it is necessary to pick specific problem types and solution procedures. For illustrative purposes a 3-d elasticity problem (3 degrees of freedom per node) using quadratic tetrahedral elements and an iterative solver that uses an assembled global stiffness matrix, stored using compressed row storage, will be considered. The total storage for the solution process will be 375 words/node (6 words/node for the solution and 369 words/node for the global stiffness matrix). If the classic data structure is used it will add another 15 words/node for a total of 390 words/node. If the largest hierarchic data structure is used (the one-level implementation) it will add 31 words/node for a total of 406 words/node. This amounts to a 4% difference in storage space. The same example using hexahedral elements results in about a 1% difference. It can be clearly seen that in this case the difference in using the richer data structure for the mesh is not significant.

8 Comparison to Special Purpose Hierarchic Data Structures

There have been some published hierarchic data structures used in adaptive analysis that, although not entirely general purpose, are well suited to the functions required by the specific procedures. This section investigates the storage penalty incurred by using the general purpose data structure described here in place of one that is specifically designed for the problem being solved. All of the data structures presented were specifically designed to handle only tetrahedral meshes which saves some storage space since the number of downward adjacencies is fixed.

8.1 Edge-Based Data Structure

Biswas and Strawn [14] present a data structure that is tailored to an edge-based analysis and refinement scheme. This data structure is a cross between the one-level adjacency structure and the reduced interior representation presented here. In their data structure interior faces are omitted but faces classified on the boundary of the model are included as indicated in Figure 18. Edges are included on both the interior and the boundary. Their data structure does not have classification information.

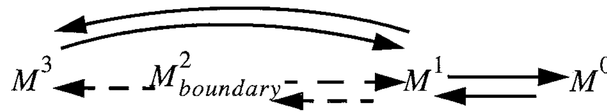


Figure 18. Data structure of Biswas and Strawn [14]

A calculation of the size of their data structure, including only the mesh information (not the solution storage which is also given in their paper) gives a storage of $22.5N_M^3$. This can be compared to the values in the first row of Table 20, which shows it to be roughly three times the classic data structure and between the circular and reduced-interior representation for the hierarchic data structures.

8.2 Data Structure with Fast Retrieval of Downward Adjacencies

Kallinderis and Vijayan present a data structure for unstructured tetrahedral meshes in Reference 13. This data structure contains all four topological mesh entities and primarily downward adjacency information as shown in Figure 19. Retrieving some adjacencies with this data structure would require global searching, however their adaptive analysis does not need these adjacencies.

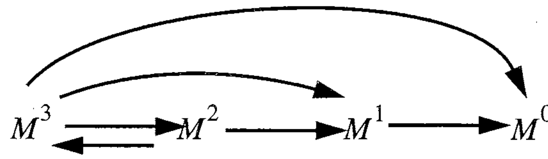


Figure 19. Data structure of Kallinderis and Vijayan [13]

It appears that this structure is optimized for speed since it explicitly stores the adjacencies required by their adaptive procedure, rather than deriving them from other adjacencies. Their data structure also does not have classification information.

The size of their data structure works out to $27N_M^3$ which is roughly the same as the circular representation of the hierarchic data structure.

8.3 Data Structure with Only Downward Adjacencies

Connell and Holmes give a data structure in Reference 15 that is structured as shown in Figure 20 providing only downward adjacencies. They do however include classification information and correctly reposition vertices classified on model boundaries during mesh refinement. Again, such a data structure would require global searching for some adjacencies, but apparently their analysis does not need these adjacencies.

Figure 20. Data structure of Connell and Holmes [15]

The storage for this representation requires $17.5N_M^3$. This is a bit more than twice the classic data structure, half of the one-level adjacency structure and between the storage for the circular and reduced-interior representations.

9 Closing Remarks

Requirements for a general purpose database based on a hierarchy of topological entities that makes up a mesh were presented. Three implementations that meet these requirements were given and compared to the classic element-node representation typically used in finite element analysis codes. It was shown that when all the storage needed for the solution process was considered (including node renumbering, if needed, and global stiffness storage), the hierarchic representation does not add a significant amount of extra storage.

The hierarchic representations add important information and capabilities that are needed for other parts of an adaptive analysis environment. These representations maintain the relation of the mesh to the geometric model that it was created from which is critical for mesh generation and enrichment procedures. Also the ability to explicitly query and manipulate all the topological entities in the mesh allows various analysis and mesh generation procedures to be written in a more natural manner. Also this representation can easily be extended to properly represent meshes of non-manifold models.

10 References

1. R. Löhner, 'Some useful data structures for the generation of unstructured grids', *Comm. in Applied Numerical Methods*, 4, 123-135 (1988).

2. J. Bonet and J. Peraire, 'An alternating digital tree (ADT) algorithm for 3D geometric and intersection problems', *IJNME*, 31, 1-17 (1991).

3. H. Dannelongue and P. Tanguy, 'Efficient data structure for adaptive remeshing with the FEM', *J. of Computational Physics*, 91, 94-109 (1990).

4. G. F. Carey, M. Sharma and K. C. Wang, 'A class of data structures for 2-D and 3-D adaptive mesh refinement', *IJNME*, 26, 2607-2622 (1988).

5. M. C. Rivara, 'Design and data structure of fully adaptive, multigrid, finite element software', *ACM Trans. Math Soft.*, 10, 242-264 (1984).

6. W. C. Rheinboldt and C. K. Mesztenyi, 'On a data structure for adaptive finite element mesh refinements', *ACM Trans. Math. Soft.*, 6, 166-187 (1980).

7. P. Devloo, J.T. Oden and T. Strouboulis, 'Implementation of an adaptive refinement technique for the SUPG algorithm', *Comp. Meth. Appl. Mech. Eng.*, 61, 339-358 (1987).

8. N. Golias and T. Tsiboukis, 'An approach to refining three-dimensional tetrahedral meshes based on Delaney transformations', *IJNME*, 37, 793-812 (1994).

9. K. C. Chellamuthu, N. Ida, 'Algorithms and data structures for 2D and 3D adaptive finite element mesh refinement', *Finite Elements in Analysis and Design*, 17, 205-229 (1994).

10. R. Löhner, 'Edges, starts, superedges and chains', *Comp. Meth. in Appl. Mech. Eng*, 111, 255-263 (1994).

11. R. Löhner, 'Some useful renumbering strategies for unstructured grids', *IJNME*, 36, 3259-3270 (1993).

12. D. M. Hawken, P. Townsend and M. F. Webster, 'The use of dynamic data structures in finite element applications', *IJNME*, 33, 1795-1811 (1992).

13. Y. Kallinderis and P. Vijayan, 'Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes', *AIAA Journal*, 31, 1440-1447 (1993).

14. R. Biswas and R. Strawn, "A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids", AIAA-93-0672, presented at the 31st Aerospace Sciences Meeting & Exhibit, Jan. 11-14, 1993, Reno, NV.

15. S. D. Connell and D. G. Holmes. '3-dimensional unstructured adaptive multigrid scheme for the euler equations', *AIAA Journal*, 32, 1626-1632 (1994).

16. O. C. Zienkiewicz and R. L. Taylor, *The Finite Element Method - Volume 1*, 4th Edition, McGraw-Hill Book Co., New York, 1987.

17. T. J. R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice Hall, Englewood Cliffs, NJ, 1987.

18. W. J. Schroeder and M. S. Shephard, 'A combined octree/delaunay method for full automatic 3-D mesh generation', *IJNME* 29, 37-55 (1990).

19. Schroeder, W. J. and Shephard, M. S. 'On rigorous conditions for automatically generated finite element meshes', In J. Turner and J. Pegna and M. Wozny, editors, *Product Modeling for Computer-Aided Design and Manufacturing*, 267-281. North Holland, 1991.

20. M.S. Shephard and M.K. Georges. 'Reliability of automatic 3D mesh generation', *Comp. Methods in Applied Mech. and Eng.*, 101, 443-462 (1992).

21. H. L. de Cougny and M. S. Shephard, 'Parallel mesh adaptation by local mesh modification', in preparation for submission, (1995).

22. M.S. Shephard, 'The specification of physical attribute information for engineering analysis', *Engineering with Computers*, 4, 145-155 (1988).

23. M. S. Shephard and P. M. Finnigan. "Toward automatic model generation." in A. K. Noor and J. T. Oden, editors, *State-of-the-Art Surveys on Computational Mechanics*, 335-366. ASME, 1989.

24. S. Dey and M. Shephard, 'Mapping finite element entities to true model geometry', Third U.S., National Congress on Computational Mechanics, Dallas, Texas, June 12-14, 1995.

25. A. A. G. Requicha and H. B. Voelcker, 'Solid modeling: Current status and research directions', *IEEE Computer Graphics and Applications*, 3, 25-37 (1983).

26. K.J. Weiler, 'The radial-edge structure: a topological representation for non-manifold geometric boundary representations', In M.J. Wozney, H.W. McLaughlin, J.L. Encarnacao, editors, *Geometric Modeling for CAD Applications*, 3-36. North Holland, 1988.

27. E. L. Gursoz, Y. Choi, and F. B. Prinz, 'Vertex-based representation of non-manifold boundaries', In M. J. Wozny, J. U. Turner and K. Priess, editors, *Geometric Modeling Product Engineering*, 107-130, North Holland, 1990.

28. K.J. Weiler, "Topological structures for geometric modeling", Ph.D. Thesis, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy NY, May 1986.

29. E. Bruzzone, L. De Floriani and E Puppo, 'Manipulating three-dimensional triangulations' in *Lecture Notes in Computer Science,* 367, 339-353, Springer-Verlag Berlin, 1989.

30. L. T. Souza and D. W. Murray, 'A unified set of resequencing algorithms', *IJMNE*, 38, 565-581 (1995).

31. S. W. Sloan and W. S. Ng, 'A direct comparison of three algorithms for reducing profile and wavefront', *Computers & Structures*, 33, 411-419 (1989).

32. I. Duff, R. Grimes, and J. Lewis, 'Sparse matrix test problems', *ACM Trans. Math. Soft.*, 15, 1-14 (1989).