# Efficient Storage of Adaptive Topological Meshes

Dan Ibanez

May 26, 2014

## 1 Mesh Data Structure

We present an efficient array-based storage structure for mesh topology and associated data which remains flexible enough to handle general unstructured mesh adaptivity. The goal of this structure is to be at least as flexible as existing object-based representations while providing the efficiency gains available from array-based storage. This storage scheme and associated algorithms will be referred to as the Mesh Data Structure, or MDS. A prototype of the MDS structure was implemented in the C language and compared to an topological structure implemented using C++ objects.

## 2 Requirements

This data structure is designed to fit the requirements laid out in the work of Shephard and Beall [1]. It represents an attempt to condense the structures described by that work into contiguous arrays rather than individual objects.

We begin by listing our requirements for a mesh data structure.

1. It must be possible to iterate through all the entities of a given type in a mesh.
2. It must be possible to compare two entities to see if they are the same.
3. First order adjacencies must be retrievable in $O(1)$ time.
4. It must be possible to uniquely associate arbitrary data with each entity.
5. Boundary edges and faces must be orientable.
6. Addition and removal of entities must be constant-time operations.

Using the data structure presented here in a full one-level configuration [1], all of these requirements are fulfilled.

# 3 Limitations

This structure is built upon a few critical assumptions. First, the mesh must be composed of entities which belong to a small set of topological types. More specifically, the number of $d$-dimensional downward adjacencies of an entity of type $t$ must be a constant function of $d$ and $t$ whose value is known at compile time. In MDS, we refer to these values as degrees, using graph theory terminology associated with a graph representation of adjacencies. $\deg(t, d)$ is the number of entities of dimension $d$ adjacent to an entity of type $t$.

General polyhedral meshes are the main exception to this requirement. Regarding polyhedral meshes, it should be possible for the MDS structure to accept new topological types at runtime, though its performance will degrade as the number of types grows large. Good performance can be found using about ten types or less, but when more than a hundred types are involved this may no longer be the optimal structure.

The other limitation of this structure relates to memory use during entity removal. In this particular design, memory use does not decrease when entities are removed. This is a deliberate design choice discussed in Section 7.2. It also takes into account the fact that most target user applications add at least as many entities as they remove.

# 4 Flexibility

Both Shephard [1] and Garimella [3] describe multiple representations for a topological mesh structure. The design of MDS is such that adjacency relations between each pair of dimensions are stored in a separate set of arrays. The structure is flexible enough include or omit adjacencies between each pair of dimensions dynamically at runtime. This allows MDS to implement all topological mesh representations.

Because MDS takes a struct-of-arrays approach as opposed to array-of-structs, the book-keeping required for flexibility is handled once at the mesh level. This is great performance advantage over designs which embed the book-keeping in entity objects, since the the combination of adjacencies stored is the same for all entities.

# 5 Entity identifiers

One of the first building blocks involved in this work is the design of an identifier which will behave the way pointers to entity objects currently do. Identifiers serve in part to satisfy requirement 2 from Section 2.

An identifier for an entity in a part must be different from all identifiers of other entities in the same part. Currently pointers are unique to their address space, so this is equivalent. An identifier for an entity must also remain the same while an entity exists on that part. Entity pointers in an object-based

representation do not change unless the entity is removed from the part, either through removal from the mesh or migration to another part.

Since the majority of our storage will be in contiguous arrays per topological type, the identifier is constructed to facilitate indexing into these arrays. An entity identifier is a pair $(t, i)$ where $t$ is a topological type and $i$ is an index unique to entities of that topological type. Although it could be a structure containing two integers, we find that it is quite convenient to encode them into a single integer. Let $T$ be the number of topological types, which is fixed at compile time. Then the equations for encoding identifiers are quite simple:

$$e = iT + t$$

$$i = \left\lfloor \frac{e}{T} \right\rfloor$$

$$t = e \mod T$$

Identifiers therefore represent on-part mesh entities uniquely and are readily convertible to and from $(t, i)$ pairs using one or two CPU integer operations. In C, such identifiers are typed `mds_id` and can be either 32-bit or 64-bit integers, depending on the maximum allowable size of a mesh part. 32-bit integers are used by default, since there is a 2X reduction in memory use and billion-element parts are rare but not unheard of.

# 6 Adjacency Structures

One of the key contributions of this work is a method of storing topological upward adjacencies in a contiguous array with minimal wasted space. This section introduces the design of these topology structures starting with examples followed by a rigorous definition.

## 6.1 Simple Meshes

To get an idea of how MDS adjacencies work, consider a simple two-triangle mesh such as the one in Figure 1.

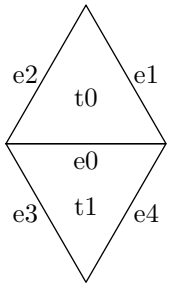


Figure 1: Two-triangle mesh

The adjacency relations between triangles and edges in this mesh may be represented by the bipartite graph shown in Figure 2.

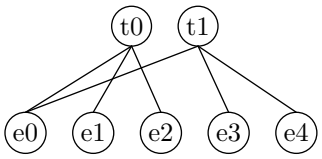


Figure 2: Adjacency graph

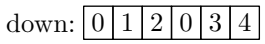


Figure 3: Downward adjacency structure

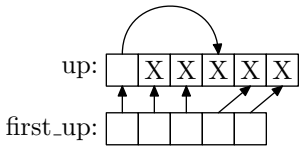


Figure 4: Upward adjacency structure

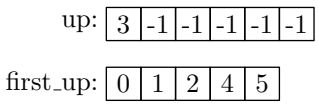


Figure 5: Upward adjacency encoding

The downward adjacency array from triangles to edges is show in Figure 3 and matches the mesh in Figure 1 (depending on orientation and order of the triangles). The corresponding upward adjacency structure from edges to triangles is shown in Figure 4.

Given a triangle t$i$, its $j$-th downward adjacency is located in `down`$[3i + j]$, where 3 is just the number of edges adjacent to a triangle. Knowing that every triangle is always adjacent to exactly three edges allows us to encode the array in this way without wasting any entries.

Notice that this array relates directly to the bipartite adjacency graph; every entry in the `down` array corresponds to an adjacency graph edge. The degree of a graph vertex representing a triangle, $\deg(t_i)$, is always 3, so there are $3n_t$ adjacency relations for $n_t$ triangles.

Now, we are concerned with storing the inverse relations from edges to triangles. We can use the fact that the total number of relations is the same ($3n_t$). As such, they can be stored in an array laid out just like the `down` array. Now,

we need a scheme for grouping together relations involving the same vertex. Although we could make them contiguous, this will make mesh modification difficult later down the line. Fortunately, we can store equivalent information by using singly-linked lists for grouping.

The array `up` is equal in size to the `down` array, and again every entry represents a triangle-edge relation. This time, the entries are singly-linked list nodes, their content is an encoded pointer to the next node. The `first_up` array has $n_e$ entries, where $n_e$ is the number of edges, and its entries are the heads of the lists of relations involving each edge. That is, the list starting at entry $i$ contains all relations involving edge $i$.

In Figure 4, an arrow from entry $a$ to entry $b$ signifies entry $a$ has a pointer to entry $b$, and an "X" in entry $a$ means that entry $a$ contains a null pointer, and is the end of a list. Figure 5 shows the same structure in terms of integer indices instead of arrows. Note that we use -1 to denote a null pointer since 0 is a valid array index.

The key to converting this lists-in-arrays structure is the fact that entry $(3i + j)$ in `up` represents the $j$-th downward relation of triangle $i$. That is, the offset of an entry in the array encodes the information about the triangle at the other end of the relation.

## 6.2 Mixed Meshes

The example illustrated in Section 6.1 is representative of meshes which are composed of a single element type. Exceptions to this rule are called mixed meshes. Notable examples include meshes containing structured boundary layers. In these cases, we have to separate arrays by type. To illustrate this complication, consider a mesh of one triangle and one quadrilateral as illustrated in Figure 6.

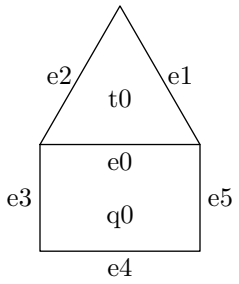


Figure 6: Simple mixed mesh

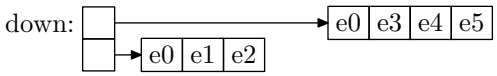


Figure 7: Mixed downward adjacency

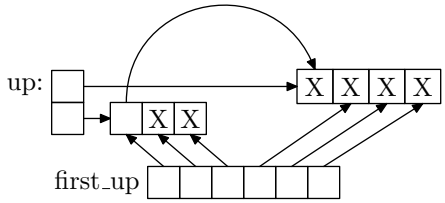


Figure 8: Mixed upward adjacency

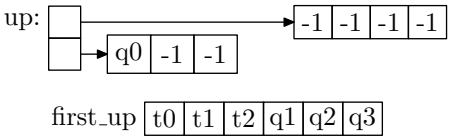


Figure 9: Mixed upward adjacency encoding

In this case, one element as three downward adjacent edges and the other has four. We group elements by topological types and use the same representation as before for each type. Figure 7 shows how the downward adjacency storage has split. We can still find the downward adjacencies for a given entity since our identifiers encode both type $t$ and index $i$, so we can pick the sub-array based on type and use $3i + j$ or $4i + j$ to index into the sub-array and find the $j$-th adjacent entity.

The same is done for upward adjacencies as shown in 8. However, this introduces an ambiguity of indexing. Is the first upward adjacent entity of an edge a triangle or quadrilateral ? This cannot be known *a priori*, so it must be encoded into the array entries. To do this, we use the same $(t, i)$ encoding introduced in Section 5, but this time $i$ is just an index into the array, and there may be multiple indices per entity. Figure 9 shows the index encoding of this structure. For example, q2 denotes the encoding $(t, 2)$, where $t$ is the arbitrary integer constant associated with the quadrilateral type.

Although not strictly necessary, downward adjacency entries also use $(t, i)$ encoding for consistency, as shown in Figure 7.

## 6.3 Final Structure

After all these considerations are made, the final declaration of the `down` array in the C language is as follows:

```
mds_id* down[4][MDS_TYPES];
```

The entity identifier for the $j$-th $d$-dimensional downward adjacency of $(t, i)$ is stored in `down`$[d][t][Ci+j]$, where $C$ is the number of $d$-dimensional downward adjacencies of all entities of type $t$. The number 4 comes from the reasonable assumption that space has 3 or fewer dimensions.

Likewise, the upward adjacency structures have the following C declarations:

```
mds_id* up[4][MDS_TYPES];
mds_id* first_up[4][MDS_TYPES];
```

`up`$[d][t]$ is the same size as `down`$[d][t]$ and contains linked-list nodes for upward adjacencies from dimension $d$ to entities of type $t$. `first_up`$[d][t]$ has one entry per entity of type $t$; that entry is the head of a list of upward adjacencies from that entity to entities of dimension $d$.

### 6.4 Storage and Lookup

Looking up the stored $d$-dimensional downward adjacency of an entity $(t, i)$ is fairly simple: just read $C$ entries starting at `down`$[d][t][Ci]$, where $C$ is the number of adjacent entities, which must be known *a priori*.

Looking up the stored $d$-dimensional upward adjacency of entity $(T, i)$ requires iterating over the list nodes starting with $(t_0, x_0)$ =`first_up`$[d][T][i]$, stopping when the constant -1 is found instead of a valid $(t_j, x_j)$ value. The node after $(t_j, x_j)$ is $(t_{j+1}, x_{j+1})$ =`up`$[D][t_j][x_j]$, where $D$ is the dimension of type $T$. Every time node $(t_j, x_j)$ is visited, that means entity $(t_j, \lfloor x_j/C_j \rfloor)$ (which has dimension $d$) is adjacent to entity $(T, i)$, where $C_j$ is the number of entities of dimension $d_l$ adjacent to an entity of type $t_j$.

The relation of $(t_0, i_0)$ being the $j$-th downward adjacency of entity $(t_1, i_1)$ is actually two relations: upward and downward. The downward relation is stored by assigning `down`$[d_0][t_1][Ci_1 + j] = (t_0, i_0)$, where $d_0$ is the dimension of type $t_0$.

The upward relation is stored by linking the node at `up`$[d_0][t_1]][Ci_1 + j]$ into the list whose head is `first_up`$[d_1][t_0][i_0]$, where $d_1$ is the dimension of type $t_1$. The node is placed as the new first node in the list, which can be done in $O(1)$ time by copying the head's value to the node and then having the head point to the node. Instead of pointer assignments, we assign $(t, i)$ encoded values to entries in `first_up` and `up` as described in the lookup algorithm.

## 7 Mesh Modification

The adjacency structures in Section 6 were presented in the context storing a description of an unchanging mesh, and are complex enough even in that case. Here we consider how they can be used efficiently to implement mesh modification operations.

Mesh modification is required by mesh adaptation, and we know from experience developing the MeshAdapt library that all the relevant mesh modification operations can be implemented as a series of mesh entity additions and removals. Addition and removal would appear to be the weak spots of an array-based representation, but this section describes how we can implement them such that performance is no worse than an object-based structure.

## 7.1 Entity Addition

Given a mesh represented by topology structures as described in 6, we need an efficient way to add an entity to the mesh. The first problem is that there is no room left in the arrays, so they must increase in size. If the size of one array is $n$, increasing it to $n + 1$ requires copying $n$ entities from the old memory to the new memory. However, we can also increase it to size $\alpha n, \alpha > 1$ with the same amount of copying. This leaves auxiliary space at the end of the array, which can be used for other entity additions, meaning that those additions happen immediately with no copying. This well-known scheme for growing arrays amortizes the cost of reallocation and results in an $O(1)$ average runtime complexity.

Notice that although the memory has moved, all entities are identified by array offsets, which have not changed, so the identifiers still obey the desired properties.

## 7.2 Entity Removal

When adding entities, we had control over their resulting index and chose to conveniently place it at the end of the active array. Removals, however, are specified by the user and may happen at any place within the array. Once the entity is removed, it leaves an unused hole in the array.

One option would be to fill this hole immediately, either by moving all subsequent entities down by one or just taking the last entity and moving it to fill the hole. Both of these approaches invalidate entity identifiers, which is bad for the user interface. In addition, the first approach makes removal too slow and the second approach changes the order of entities. For these reasons, we choose to let a hole remain after removal.

We track these holes with one array-based linked list per topological type. Entries in the array are linked list nodes, and there is an external head pointing to the first hole. The remaining holes are linked together. Entries which do not represent holes store a "live" constant which is distinguishable from valid and null pointer values.

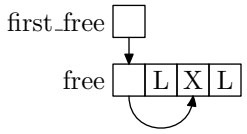


Figure 10: Free list after two removals

Figure 10 shows a free list for a topological type which began with 4 entities and had indices 2 and 0 removed, leaving two holes.

The purpose of tracking these holes is to allow subsequent entity additions to fill them. When the user requests an addition, this free list is first checked. If there are holes in the free list, the first one is unlinked and its index is used

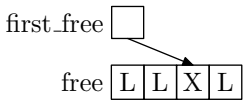


Figure 11: Free list after an addition

for the new entity. Figure 11 illustrates the free list after an entity addition filled the hole at index 0. This algorithm ensures that all holes are filled before entities are added at the end of the array.

### 7.3 Range Markers

The ability to handle additions efficiently requires allocating some extra space at the end of arrays for future additions to make use of. This separates the array into two parts: the part that is actually used and the unused part at the end. Entity removal, in turn, allows holes to be made in the used part of the array, which means the number of "live" entries is not the same as the number of "used" entries.

To keep track of all this, three integers are maintained for each topological type. The `n` counter keeps track of the actual number of entities of that topological type currently in the mesh part. This should be equal to the number of used entries minus the number of holes. The position of the first unused entry in the array is kept in `end`, which is also the number of used entries. Finally, the actual allocated size is stored in `cap`.

## 8 Results

The following table lists mesh memory usage (reported as "in use bytes" by the C/C++ memory allocator) for different mesh libraries when loading a serial all-tet mesh of 93710 elements. The mesh has matched entities along the surface. All of these libraries are storing the same information: a full one-level adjacency representation, coordinates, classification, matches, etc.

The first two are similar C++ object based structures, "pumi" being the version developed at SCOREC and "sim" at Simmetrix, Inc.

| library | MB | bytes per tet | relative |
|---|---|---|---|
| pumi | 84.1 | 897 | 100% |
| sim | 55.2 | 589 | 66% |
| mds | 22.6 | 241 | 27% |

Simmetrix also offers a reduced representation structure. Re-generating a mesh of similar size in Simmetrix reduced form uses about 245 bytes per element, almost exactly what MDS uses for a full representation. Note that both libraries are storing additional information such as coordinates and classification.

If we remove from MDS all the non-topological data such as coordinates, classification, etc., we are left with the core MDS structure described above, which uses 12.5 MB of data for this mesh, or 133 bytes per element.

In addition, we can predict the memory use of an application using a reduced representation based on MDS by removing all interior faces and edges, then storing only the one-level adjacencies between elements and vertices, faces and edges, and edges and vertices. That structure took up 7.4 MB for this mesh. This works out to about 79 bytes per element for a reduced representation in MDS.

By comparison, Dyedov et al published a half-facet mesh data structure called AHF, implemented both within the MOAB library and as standalone C code generated from MATLAB code [2]. This structure stored only the topological data, and memory use was measured for both implementations on various meshes. For their largest mesh, which had similar proportions of explicit faces to our test mesh, the MOAB_AHF implementation used 80 bytes per element while the standalone integer-based AHF C code used about 39 bytes per element.

# References


[1] Mark W Beall and Mark S Shephard. Mesh data structures for advanced finite element applications. *SCOREC Report*, pages 23–1995, 1995.

[2] Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and Timothy J Tautges. Ahf: Array-based half-facet data structure for mixed-dimensional and non-manifold meshes. In *Proceedings of the 22nd International Meshing Roundtable*, pages 445–464. Springer, 2014.

[3] Rao V Garimella. Mesh data structure selection for mesh generation and fea applications. *International journal for numerical methods in engineering*, 55(4):451–478, 2002.