Array-based Half-Facet (AHF) Data Structure for Mixed-Dimensional Non-manifold Meshes

March 4, 2013

1 Introduction

The design objectives of our data structure include the following:

Conceptual simplicity. It should be simple and intuitive, with a minimum number of concepts.

Coherence. The data structure should be coherent for different dimensions.

Computational efficiency. It should support all local adjacency queries in constant time, assuming the valence of each entity is bounded by a small constant.

Minimal extra storage. It should require minimum amount of storage additional to the element connectivity (i.e., list of nodal IDs of each entity), which are stored explicitly in general.

The basic concept of our data structure is a *half-facet* of an entity, which is an oriented $(d-1)$st dimensional sub-entity of a d-dimensional entity. In particular, a half-facet of a cell is a *half-face*, a half-facet of a face is a *half-edge*, and a half-facet of an edge is a *half-vertex*.

In a manifold mesh, a facet can have up to two half-facets (with opposite orientations), which are commonly referred to as *twin* half-facets. A boundary half-facet does not have a twin half-facet. In a non-manifold mesh, a facet can have more than two half-facets, which we refer to as *sibling half-facets*.

Using the the concepts of *half-facet* and *sibling half-facets*, we can represent non-manifold meshes in 1-D, 2-D, or 3-D. In particular, besides the vertex coordinates and the element connectivity of a d-dimensional mesh, we store 1) the mapping of sibling half-facets and 2) the mapping from a vertex to a half-facet.

For a non-manifold meshes with mixed dimensional entities, we use the union of the non-manifold meshes of the different dimensions. Assuming that the valence of each entity is bounded by a constant, this union will allow perform any top-down or bottom-up query in constant time. For better efficiency, some auxiliary arrays for intermediate-dimensional entities (i.e., edges and faces of a volume mesh) may be constructed and cached dynamically to speed up the operations. Such a caching strategy is effective and efficient, because in general only a small subset of the intermediate-dimensional entities (i.e., edges and faces of a volume mesh) are stored explicitly, such as those incident on the boundary or internal boundaries.

2 Design and Implementation

Figure 1 illustrates the design of the AHF for a non-manifold mesh with mixed-dimensional entities. Our data structure is composed of the following three core parts:

1. Half-face data structure for 3-D entities, composed of three arrays:
   (a) cells: a list of vertex IDs for each cell;
   (b) sibhf: a map from each half-face to a sibling half-face (the map forms a loop for the sibling half-faces corresponding to the same face);
   (c) v2hf: a map from a vertex to a half-face ID.

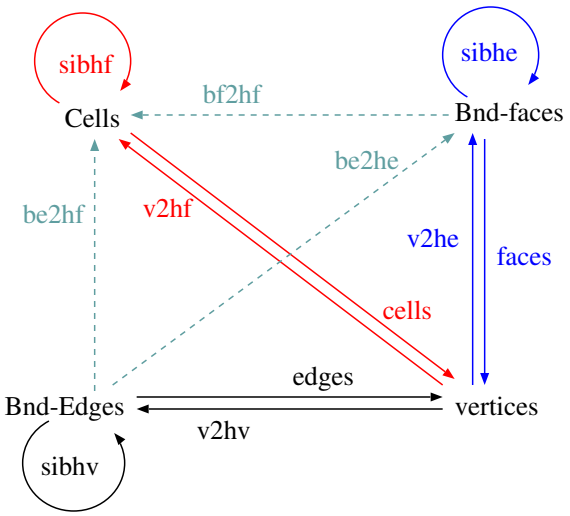

Figure 1: Design of AHF.

2. Half-edge data structure for 2-D entities, composed of three arrays:
   (a) faces: a list of vertex IDs for each (boundary) face;
   (b) sibhe: a map from each half-edge to a sibling half-edge (the map forms a loop for the sibling half-edges corresponding to the same edge);
   (c) v2he: a map from a vertex to a half-edge ID.
3. Half-vertex data structure for 1-D entities, composed of three arrays:
   (a) edges: a list of vertex IDs for each (boundary) edge;
   (b) sibhv: a map from each half-vertex to a sibling half-vertex (the map forms a loop for the sibling half-vertices corresponding to the same vertex);
   (c) v2hv: a map from a vertex to a half-vertex ID.

In addition, we have some optional auxiliary cache for intermediate-dimensional entities:

1. bf2hf: boundary face ID to a half-face ID;
2. be2hf: boundary edge ID to a half-face ID;
3. be2he: boundary edge ID to a boundary half-edge ID.

Each entry in these arrays can be computed in constant time, so we don't need to precomputed them in a batch mode, and instead can compute and cache them on the fly.