Chapter 5

Mesh data structures

As has been mentioned in Section 3.4, in addition to standard mesh structures it is necessary in higher-order FEM to maintain vertex, edge and bubble *nodes*, from which one can construct connectivity arrays.

The need for mesh adaptivity brings further demands that must be taken into account in the design of the data structures. A tree-like hiearchy of element refinement levels must be introduced and also more complicated neighbor search algorithms are required to correctly initialize the edge and vertex nodes. The complexity of the data structures reflects strongly on the complexity of the rest of the finite element code.

In the following we present an original design of data structures for adaptive meshes, free of any regularity assumptions yet retainign extreme simplicity.

5.1 Node and element structures

It should be noted first that we have departed from traditional mesh data structures storing complete information about the finite element problem. Our mesh structures contain geometrical information only – the remaining data, including basis function indices, boundary conditions, polynomial degrees, etc., are stored in separate classes describing concrete finite element spaces (H^1, $\boldsymbol{H}$(curl), ...) and are accessible via the `id` numbers of nodes and elements. This was done to allow multiple spaces and multiple element

types to coexist on the same mesh, which is vital for solving multi-physics and coupled problems.

The entire mesh is defined by two arrays of the following C structures. The first one stores all nodes:

```
struct Node
{
  // int id; (implied)
  unsigned ref:30;
  unsigned used:1;
  unsigned type:1;

  union {
    struct { // vertex node data
      double x, y;
    };
    struct { // edge node data
      int marker;
      Element* elem[2];
    };
  };

  int p1_id, p2_id;
  Node* next_hash;
};
```

This structure defines both vertex and edge nodes by utilizig the `union` construct. The standard vertex positions `x`, `y`, while typically stored separately, were placed in the vertex variant of the `Node` structure for simplicity. The edge variant contains an edge marker (used for identifying different boundaries of the computational domain) and pointers to the two elements sharing the edge node (useful, e.g., when enforcing the minimum rule for edge polynomial degree).

The `id` number, pointing to extra node data in one or more FE space tables, is implied from the position of the structure in the array. Note that the size of the structure is 32 bytes, thus the division involved in the calculation of `id` can be performed by a bit shift.

The members `type` and `used` determine the variant of the structure and whether the particular item of the node array is used, respectively. The remaining members will be described later.

Elements are defined by the second structure:

```
struct Element
{
  int id;
  unsigned marker : 29;
  unsigned active : 1;
  unsigned used : 1;
  unsigned type : 1;

  Node* vn[4]; // vertex nodes
  union {
    Node* en[4]; // edge nodes
    Element* sons[4]; // son elements
  };
};
```

An element can be either active or inactive, hence the one-bit variable `active`. Active elements store pointers to the appropriate vertex and edge nodes. Inactive elements are those which have been split and are excluded from the computation. Their purpose is to hold pointers to the descendant elements. The constraint update algorithm requires the inactive elements to preserve the vertex node pointers, which is why the array `vn` is outside the union.

Triangular and quadrilateral elements share the same structure and are distinguished by the member `type`. The fourth vertex node pointer is unused for triangles, but this is worth the simpler code that results from the shared structure. The rest of the variables are analogous to the `Node` structure.

5.2 Eliminating neighbor search by hashing

To properly initialize edge node pointers after reading a mesh file, one has to construct neighbor lists for all elements and use them in such a way that only one node is created for each edge. Further problems arise when certain elements are refined after automatic mesh adaptation. Unless hanging nodes are removed by extra refinements, no longer is each edge shared by at most two elements. Standard neighbor lists fail to fully capture such situations and thus more complex data structures, e.g., trees [12], have to be introduced.

We have avoided all of these problems by introducing a function which, given the `id` numbers of two nodes, returns a pointer to the node halfway between

them. If no such node exists, it is created first. The task of translating two numbers to a node pointer is accomplished using a hash table.

We are maintaining two independent layers of nodes: the first layer contains all vertex nodes, the second all edge nodes. The following two functions can be called:

```
Node* get_vertex_node(int id1, int id2);
Node* get_edge_node(int id1, int id2);
```

All nodes, apart from being accessible by their `id` number, can be reached using these functions by passing the `id`s of their "parent" nodes. Top-level vertex nodes (those loaded from the mesh file) are stored at the beginning of the node array and can be accessed directly without hashing. Mesh initialization then becomes trivial:

```
nodes = // read all top-level vertex nodes from a file
for (all e in elements) {
  vv[3] = // read element vertex id numbers
  for (i = 0; i < 3; i++) {
    e->vn[i] = &nodes[vv[i]];
    e->en[i] = get_edge_node(vv[i], vv[(i+1)%3]);
  }
}
```

Element refinement is also very straightforward. No care must be taken of the neighboring elements, regardless of their refinement level or even existence:

```
Element* create_triangle(Node* v0, Node* v1, Node* v2)
{
  Element* e = new Element;
  e->active = 1; e->type = 0; // etc.
  e->vn[0] = v0; e->vn[1] = v1; e->vn[2] = v2;
  e->en[0] = get_edge_node(v0->id, v1->id);
  e->en[1] = get_edge_node(v1->id, v2->id);
  e->en[2] = get_edge_node(v2->id, v0->id);
  // reference all nodes of the new element
  return e;
}

void refine_element(Element* e)
{
  Node* x0 = get_vertex_node(e->vn[0]->id, e->vn[1]->id);
  Node* x1 = get_vertex_node(e->vn[1]->id, e->vn[2]->id);
  Node* x2 = get_vertex_node(e->vn[2]->id, e->vn[0]->id);
```

```
    e->sons[0] = create_triangle(e->vn[0], x0, x2);
    e->sons[1] = create_triangle(x0, e->vn[1], x1);
    e->sons[2] = create_triangle(x2, x1, e->vn[2]);
    e->sons[3] = create_triangle(x0, x1, x2);
    e->active = 0;
    // un-reference all nodes of e (to be explained)
  }
```

Each hash table is implemented as an array of linked lists of hash synonyms (*open hash*). This hash table organization has the advantage of simple node removal, which is required if a node is no longer needed by any element. Synonym nodes are linked by the pointer `next_hash` and are distinguished by the parent node numbers `p1_id` and `p2_id`. To ensure that `get_*_node(id1, id2)` gives the same result as `get_*_node(id2, id1)`, the smaller parent `id` is always stored in `p1_id` and each query is modified accordingly as well.

There are two important parameters in the design of any hash table: the size of the table and the choice of the hash function. To prevent the table from becoming overfilled we set its size to about one half of the expected number of nodes when loading the mesh. The size is not required to be a prime number for this type of hash table. We always choose it to be a power of two to avoid the modulo operation in the hash function. A satisfactory spreading of the table items is achieved by the following hash function:

```
int hash(int id1, int id2)
  { return (A*id1 + B*id2) & (table_size-1); }
```

where `A` and `B` are large integer constants. The number of synonyms in each non-empty list is then two on average and very scarcely greater than four, which outperforms most tree representations (tested on a mesh with around one million nodes).

5.3 Determining node type

Figure 5.1 shows a simple mesh with hanging nodes. Five types of nodes can be identified: standard vertex nodes (A), standard edge nodes (B), constrained vertex nodes (C), constrained edge nodes (D) and finally standard edge nodes and constrained vertex nodes at the same place (E).

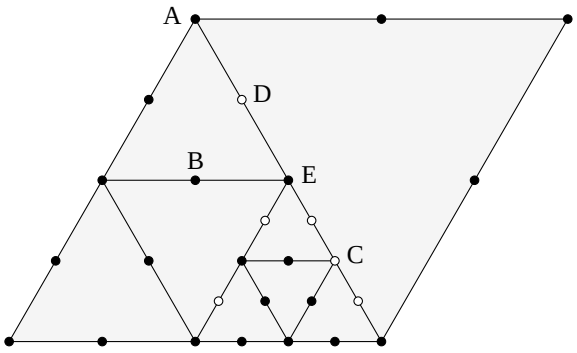

Figure 5.1: Mesh with hanging nodes.

As the elements in the mesh get refined (or un-refined), some nodes are created, others are removed and some change their types. One has to be able to quickly recognize whether a node is constrained or unconstrained without searching in the element hierarchy. This is achieved through the member `ref` of the structure `Node`. At all times this variable holds the number of elements pointing to the node. This is the reason why all nodes of an element must be *referenced* (`ref` increased) when creating the element and *un-referenced* (`ref` decreased) when the element is being refined or destroyed, as shown in Section 5.2.

Now the cases (A) and (C) can be distinguished just by looking at `ref`. If `ref` $= 3$, the vertex node is constrained, if `ref` > 3 it is not constrained. Top-level vertex nodes have `ref` artificially increased to a large number, which ensures that they are always treated as unconstrained. Similarly, if `ref` $= 1$ for an edge node, it is constrained (D); if `ref` $= 2$ it is unconstrained (D).

The case (E) can be detected by calling the function `peek_*_node`, which works the same way as `get_*_node`, with the exception that the node is not created if it does not exist. The case (E) is important, since it is a starting point for the construction of the extended connectivity lists in the recursive algorithm described in Section 4.2.3. It is also the reason for maintaining two separate layers of vertex and edge nodes.

A node is destroyed any time its `ref` reaches zero.