Algorithm 3.2 CELL NEIGHBOR SEARCH: Find cell neighbors from facet vertex sets

IN: A grid $\mathcal{G}$

OUT: A mapping $\mathcal{I} : \mathcal{G}^d \times \mathbb{N} \mapsto \mathcal{G}^d$ which maps each cell to its sequence of neighbors, such that the nth facet of a cell c is incident to the neighbor in $\mathcal{I}(c, n)$.

```
1:  $\mathcal{N} : \mathbb{P}\mathcal{G}^0 \mapsto \mathcal{G}^d \times \mathbb{N}$ is a mapping from vertex sets to (cell,local side) pairs.
2:  $\mathcal{N} \leftarrow \emptyset$
3:  for all cells $C \in \mathcal{G}^d$ do
4:    for all facets $F \prec C$ do
5:      $f \leftarrow$ local number of $F$ in $C$
6:      if $F^0 \notin \mathrm{dom}\,\mathcal{N}$ then
7:        $\mathcal{N}(F^0) \leftarrow (C, f)$   (Store $C$ and local facet number)
8:      else   (facet already found from the other neighbor $D$.)
9:        $(D, f_D) \leftarrow \mathcal{N}(F^0)$
10:       $\mathcal{N}(F^0) \leftarrow \emptyset$
11:         $\mathcal{I}(C, f_C) \leftarrow D$
            $\mathcal{I}(D, f_D) \leftarrow C$
12:     end if
13:  (Facets still in $\mathcal{N}$ are boundary facets.)
```

5. access to the vertex set of a facet, (iteration over the vertices of a facet will suffice)

6. comparison of vertex sets for equality, this implies: vertices must be equality comparable

7. in order to guarantee an efficient implementation of the map $\mathcal{N}$, such vertex sets must be either be totally ordered or endowed with a hash function, which essentially means the same requirement for vertices.

It would also be advantageous (in favor of memory efficiency) to have a sort of minimal representation for cells and vertices, given a fixed underlying grid $\mathcal{G}$.

3.2.3 Bandwidth-reducing Ordering of Grid Elements

Often, grid elements correspond to the unknowns of linear systems of equations. For solution algorithms based on elimination strategies, it is often advantageous to reduce the *bandwidth* of the matrix, that is, the maximal difference in the numbering of two unknowns coupled by a linear equation, see [Saa96].

This problem is best formulated in terms of graphs, where nodes correspond to unknowns, and an edge between two nodes means that the corresponding unknowns are connected by a non-zero entry in the matrix. The task is then to number nodes such that the maximal difference of adjacent node numbers is minimized.

A well-known heuristic for band-width minimization is the CUTHILL-MCKEE algorithm, which essentially is a breadth-first traversal which sorts each layer in order of increasing node degree.

Algorithm 3.3 CUTHILL-MCKEE ORDERING: Bandwidth minimization

IN: a graph $\mathcal{G}$ and a starting node $v_0 \in \mathcal{G}$

OUT: a numbering $N : V(\mathcal{G}) \mapsto \mathbb{N}$ *($V(\mathcal{G})$ is the vertex set of $\mathcal{G}$)*

```
 1: $n \leftarrow 1$   (current vertex number)
 2: $N(v_0) = n$
 3: $L_0 \leftarrow \{v_0\}$   (Level 0 consists of $v_0$)
 4: Mark$(v) = 0 \; \forall v \in \mathcal{G}$
 5: Mark$(v_0) = 1$   (mark $v_0$ as visited)
 6: $k \leftarrow 0$   (current level is 0)
 7: while $L_k \neq \emptyset$ do   ($L_k = \emptyset$: no new vertices found, stop.)
 8:   $L_{k+1} \leftarrow \emptyset$
 9:   for all $v \in L_k$ do
10:     if Mark$(v) = 0$ then
11:       Mark$(v) = 1$
12:       Add $v$ to $L_{k+1}$
13:       $n \leftarrow n + 1$, $N(v) \leftarrow n$
14:     end if
15:   sort $L_{k+1}$ in order of increasing degree
16:   $k \leftarrow k + 1$
17: end while
```

Conforming to the spirit of generic programming, this algorithm should be implemented in terms of graphs, rather than of grids. It is evident how this increases reuse possibilities: If we have a grid with unknowns living on, say, cells, and two unknowns connected by an equation if and only if their corresponding cells are neighbors then this grid can be mapped to such a graph quite easily, by viewing cells as nodes and facets as edges. But this is by no means the only possible way a graph can arise from grids: For example, if unknowns live on vertices, a different graph results.

Thus, it is very important to find the 'natural' mathematical structure for an algorithm to operate on — a description based on grids would have to distinguish all these cases, whereas a graph-based one does not.

Now, in order to analyze CUTHILL-MCKEE ORDERING in terms of grid functionality, we have to fix one interpretation. For simplicity, we choose the cell-facet case, afterwards pointing out what changes when taking a different case.

1. Iteration over the neighbors of cells

2. Query on the number of neighbors of a cell

3. Association of integer (N) and boolean (Mark) data to cells

A special remark applies to the mapping Mark: It would be advantageous if the assignment Mark$(v) = 0 \, \forall v \in \mathcal{G}$ could be done in constant time, especially if the algorithm is executed only on a part of the grid (graph).

If we store unknowns on vertices, and for example define the linear system such that vertices belonging to a common cell are coupled by an equation (typical for FEM Matrices), then we would have to replace ‘cell’ with vertex, and neighbor-iteration with a somewhat more complicated iteration over all cell incident to a vertex, and all vertices incident to cell; in fact, what is needed here is iteration over the *link* of a vertex (see page 48).

Things get worse when unknowns are coupled across wider distances than in this example. For typical numerical applications, this coupling can be described by so-called *stencils*, a topic examined in section 5.5.4.

The mapping of grids to graphs can be done in a generic fashion. Currently, a preliminary mapping of the cell-neighbor-graph to the GGCL libary (p. 44) is implemented. Thus, there has to be only *one* software component for each type of graph arising from grids, and one implementing the actual algorithm.

3.2.4 Graphical Output

Graphical output of grids certainly is an important tool, for example, it can give visual control over *grid quality*. Actually, the task of producing visual output consists of two sub-tasks: First, generation of geometrical entities from the grid data structures, and second, *rendering* of these entities on some graphical device, such as pixel graphics or postscript file. The rendering step also includes things such as clipping, illumination and camera positioning (for three-dimensional objects).

Here, we are only concerned with the first step, relying on a generic `draw` method of some unspecified graphics device which works for *simple* objects like lines and polygons.

The possibilities of graphical output are manifold, we confine ourselves to give some typical examples. A simple line graphic (‘wire frame’) can be produced as follows:

```
for all edges $e \in \mathcal{G}^1$ do
    GraphicsDevice.draw(segment($e$))
```

A somewhat more advanced visualization method, especially for 3D grids, is a shrink-view of cells: Each cell is shrinked by some factor towards its center, offering better insight into the connectivity:

```
IN: shrink-factor $\alpha < 1$
1: for all cells $c \in \mathcal{G}^d$ do
2:     $p_c \leftarrow$ center of $c$
3:     for all facets $f$ of $c$ do
4:         let $P$ be the polygon defined by the vertices $v$ of $f$, with vertex coordinates $x'(v) =$
           $p_c + \alpha(x(v) - p_c)$.
5:         GraphicsDevice.draw($P$)
```

The requirements of these tiny algorithms are the following:

1. iteration over all edges or over all cells
2. iteration over all facets of a cell