Chapter 5

Plane Graphs and the DCEL

So far we have been talking about geometric structures such as triangulations of polygons and arrangements of line segments without paying much attention to how to represent these structures. This will change now. As all of these structures can be regarded as plane graphs, we start by reviewing a bit of terminology from (planar) graph theory. We assume that this is—at least somewhat—familiar ground for you. If you would like to see more details and examples, please refer to any standard textbook on graph theory, such as Bondy and Murty [2], Diestel [3], or West [10].

A (simple undirected) *graph* is a pair $G = (V, E)$ where V is a finite set of *vertices* and E is the set of *edges*, $E \subseteq \binom{V}{2} := \{\{u, v\} \mid u, v \in V, u \neq v\}$. For brevity, one often writes uv rather than $\{u, v\}$ to denote an edge. The two vertices defining an edge are *adjacent* to each other and they are *incident* to the edge.

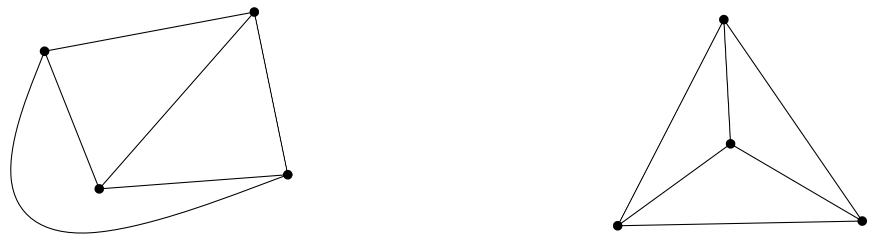
For a vertex $v \in V$, denote by $N_G(v)$ the *neighborhood* of v in G, that is, the set of vertices from G that are adjacent to v. Similarly, for a set $W \subset V$ of vertices define $N_G(W) := \bigcup_{w \in W} N_G(w)$. The *degree* $\deg_G(v)$ of a vertex $v \in V$ is the size of its neighborhood, that is, the number of edges from E incident to v. The subscript is often omitted when it is clear to which graph it refers to. As every edge is incident to exactly two vertices, we have the following simple but useful relationship, often called *handshaking-lemma*: $\sum_{v \in V} \deg(v) = 2|E|$.

A *walk* in a graph G is a sequence $W = (v_1, \ldots, v_k)$, $k \in \mathbb{N}$, of vertices such that v_i and v_{i+1} are adjacent in G, for all $1 \leqslant i < k$. A *path* is a walk whose vertices are pairwise distinct. A *cycle* is a walk whose vertices are pairwise distinct, except for $v_1 = v_k$. A graph is *connected*, if there is a path between any pair of vertices. If a graph is not connected, it is *disconnected*. A disconnected graph can be decomposed into maximal connected subgraphs, its (connected) *components*.

A *tree* is a connected graph that does not have any cycle. It is not hard to show that trees on n vertices are exactly the graphs on n vertices that have $n - 1$ edges.

In geometry we are typically concerned with graphs that are embedded on some surface, here $\mathbb{R}^2$. An *embedding* or *drawing* is just a "nice" mapping of a graph G into the plane. More formally, each vertex v is mapped to a distinct point $\phi(v) \in \mathbb{R}^2$ (that is, $\phi : V \to \mathbb{R}^2$ is injective) and each edge uv is mapped to an *arc*—a simple Jordan curve—$\phi(uv) \subset \mathbb{R}^2$ from $\phi(u)$ to $\phi(v)$ such that no vertex is mapped to the relative interior of $\phi(uv)$.

A graph is *planar* if it admits a *crossing-free* drawing, that is, a drawing in which no two arcs intersect except at common endpoints. For example, K_4 (the complete graph on four vertices) is planar, as demonstrated by the drawing shown in Figure 5.1a.



(a) Crossing-free drawing. **(b)** Straight-line drawing.

Figure 5.1: *Embeddings of* K_4 *into the plane.*

If a graph is planar, then by Fáry-Wagner's Theorem [4, 8] there also exists a *straight-line* drawing, that is, a drawing in which all arcs are line segments. In order to obtain such a drawing for K_4, we have to put one vertex inside the convex hull of the other three, see Figure 5.1b.

Sometimes we refer to graphs not as abstract graphs but in a concrete embedding. If this embedding is a crossing-free embedding into the plane, the embedded graph is referred to as a *plane graph*. Note the distinction between "planar" and "plane": The former refers to an abstract graph and expresses the possibility of a crossing-free drawing, whereas the latter refers to a geometric object that is a concrete crossing-free drawing of a graph in the plane.

A *plane straight-line graph* (PSLG) is a crossing-free straight-line drawing of a graph in the plane. Both graphs in Figure 5.1 are plane, but only the one shown on the right is also a plane straight-line graph.

5.1 The Euler Formula

If we remove all vertices (points) and edges (arcs) of a plane graph G from $\mathbb{R}^2$, then what remains is a finite collection of open sets. These sets are referred to as the *faces* of G. For instance, both plane graphs in Figure 5.1 have 4 vertices, 6 edges and 4 faces (one of which is unbounded). In general, if $|V|$ is the number of vertices of a connected plane graph, $|E|$ its number of edges and $|F|$ the number of faces, then the Euler Formula states that

$$|V| - |E| + |F| = 2.$$

In the example, we get $4 - 6 + 4 = 2$.

If you do not insist on being too formal, the proof is simple and works by induction on the number of edges. If we fix the number of vertices, the base case occurs for $|V| - 1$

edges where the plane graph is a tree. Then we have $|F| = 1$ and the formula holds. A graph with more edges always contains a cycle and therefore at least one bounded face. Choose one edge from a bounded face and remove it. The resulting graph is still connected and has one edge less but also one face less since the edge removal merges the bounded face with another one. Consequently, since the Euler Formula holds for the smaller graph by induction, it also holds for the larger graph.

The Euler Formula can be used to prove the following important fact about planar graphs.

Lemma 5.1 *Any planar graph on $n \geqslant 3$ vertices has at most $3n-6$ edges and at most $2n-4$ faces.*

This lemma shows that the overall complexity—the sum of the number of vertices, edges, and faces—of a planar graph is linear in n. A planar graph with a maximal number $(3n-6)$ of edges is called *maximal planar*.

Exercise 5.2 *Prove Lemma 5.1 using the Euler formula.*

Exercise 5.3 *Prove that every planar graph has a vertex of degree at most* 5.

Exercise 5.4 *Show that K_5 (the complete graph on five vertices) is not planar.*

5.2 The Doubly-Connected Edge List

Many algorithms in computational geometry work with plane graphs. The *doubly-connected edge list* (DCEL) is a data structure to represent a plane graph in such a way that it is easy to traverse and to manipulate. In order to avoid unnecessary complications, let us discuss only connected graphs here that contain at least two vertices. It is not hard to extend the data structure to cover all plane graphs. For simplicity we also assume that we deal with a straight-line embedding and so the geometry of edges is defined by the mapping of their endpoints already. For more general embeddings, the geometric description of edges has to be stored in addition.

The main building block of a DCEL is a list of *halfedges*. Every actual edge is represented by two halfedges going in opposite direction, and these are called *twins*, see Figure 5.2. Along the boundary of each face, halfedges are oriented counterclockwise.

A DCEL stores a list of halfedges, a list of vertices, and a list of faces. These lists are unordered but interconnected by various pointers. A vertex v stores a pointer $\mathsf{halfedge}(v)$ to an arbitrary halfedge originating from v. Every vertex also knows its coordinates, that is, the point $\mathsf{point}(v)$ it is mapped to in the represented embedding. A face f stores a pointer $\mathsf{halfedge}(f)$ to an arbitrary halfedge within the face. A halfedge h stores *five* pointers:

- a pointer $\mathsf{target}(h)$ to its target vertex,

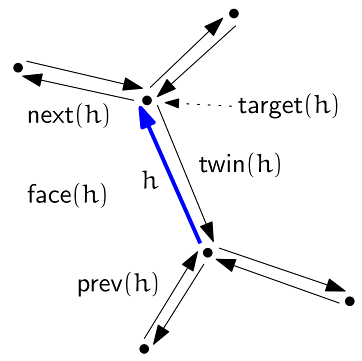


Figure 5.2: *A halfedge in a DCEL.*

- a pointer face(h) to the incident face,
- a pointer twin(h) to its twin halfedge,
- a pointer next(h) to the halfedge following h along the boundary of face(h), and
- a pointer prev(h) to the halfedge preceding h along the boundary of face(h).

A constant amount of information is stored for every vertex, (half-)edge, and face of the graph. Therefore the whole DCEL needs storage proportional to $|V| + |E| + |F|$, which is $O(n)$ for a plane graph with n vertices by Lemma 5.1.

This information is sufficient for most tasks. For example, traversing all edges around a face f can be done as follows:

```
s ← halfedge(f)
h ← s
do
    something with h
    h ← next(h)
while h ≠ s
```

Exercise 5.5 *Give pseudocode to traverse all edges incident to a given vertex v of a DCEL.*

Exercise 5.6 *Why is the previous halfedge* prev($\cdot$) *stored explicitly and the source vertex of a halfedge is not?*

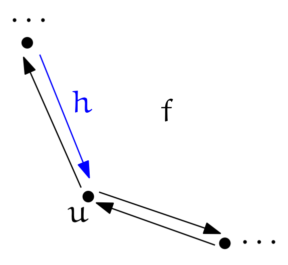
5.2.1 Manipulating a DCEL

In many geometric algorithms, plane graphs appear not just as static objects but rather they evolve over the course of the algorithm. Therefore the data structure used to represent the graph must allow for efficient update operations to change it.

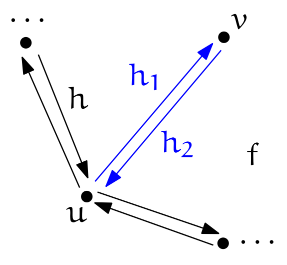
First of all, we need to be able to generate new vertices, edges, and faces, to be added to the corresponding list within the DCEL and—symmetrically—the ability to delete an existing entity. Then it should be easy to add a new vertex v to the graph within some face f. As we maintain a connected graph, we better link the new vertex to somewhere, say, to an existing vertex u. For such a connection to be possible, we require that the open line segment uv lies completely in f.

Of course, two halfedges are to be added connecting u and v. But where exactly? Given that from a vertex and from a face only some arbitrary halfedge is directly accessible, it turns out convenient to use a halfedge in the interface. Let h denote the halfedge incident to f for which $\mathsf{target}(h) = u$. Our operation becomes then (see also Figure 5.3)

add-vertex-at(v, h)
Precondition: the open line segment $\overline{\mathsf{point}(v)\mathsf{point}(u)}$, where $u := \mathsf{target}(h)$, lies completely in $f := \mathsf{face}(h)$.
Postcondition: a new vertex v has been inserted into f, connected by an edge to u.



(a) before



(b) after

Figure 5.3: *Add a new vertex connected to an existing vertex* u.

and it can be realized by manipulating a constant number of pointers as follows.

```
add-vertex-at(v, h) {
    $h_1 \leftarrow$ a new halfedge
    $h_2 \leftarrow$ a new halfedge
    halfedge$(v) \leftarrow h_2$
    twin$(h_1) \leftarrow h_2$
    twin$(h_2) \leftarrow h_1$
    target$(h_1) \leftarrow v$
    target$(h_2) \leftarrow u$
    face$(h_1) \leftarrow f$
    face$(h_2) \leftarrow f$
    next$(h_1) \leftarrow h_2$
    next$(h_2) \leftarrow$ next$(h)$
    prev$(h_1) \leftarrow h$
```

```
        prev$(h_2) \leftarrow h_1$
        next$(h) \leftarrow h_1$
        prev$($next$(h_2)) \leftarrow h_2$
    }
```

Similarly, it should be possible to add an edge between two existing vertices u and v, provided the open line segment uv lies completely within a face f of the graph, see Figure 5.4. Since such an edge insertion splits f into two faces, the operation is called *split-face*. Again we use the halfedge h that is incident to f and for which target$(h) = u$.

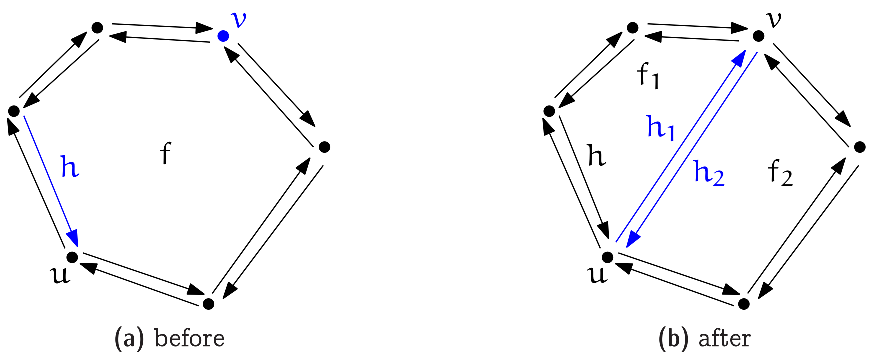


Figure 5.4: *Split a face by an edge* uv.

Our operation becomes then

```
    split-face$(h, v)$
    Precondition: $v$ is incident to $f :=$ face$(h)$ but not adjacent to $u :=$ target$(h)$.
        The open line segment $\overline{\text{point}(v)\text{point}(u)}$ lies completely in $f$.
    Postcondition: $f$ has been split by a new edge $uv$.
```

The implementation is slightly more complicated compared to add-vertex-at above, because the face f is destroyed and so we have to update the face information of all incident halfedges. In particular, this is not a constant time operation, but its time complexity is proportional to the size of f.

```
    split-face$(h, v)$ {
        $f_1 \leftarrow$ a new face
        $f_2 \leftarrow$ a new face
        $h_1 \leftarrow$ a new halfedge
        $h_2 \leftarrow$ a new halfedge
        halfedge$(f_1) \leftarrow h_1$
        halfedge$(f_2) \leftarrow h_2$
        twin$(h_1) \leftarrow h_2$
        twin$(h_2) \leftarrow h_1$
        target$(h_1) \leftarrow v$
        target($h_2$) ← $u$
        next($h_2$) ← next($h$)
        prev(next($h_2$)) ← $h_2$
        prev($h_1$) ← $h$
        next($h$) ← $h_1$
        $i$ ← $h_2$
        loop
            face($i$) ← $f_2$
            if target($i$) = $v$ break the loop
            $i$ ← next($i$)
        endloop
        next($h_1$) ← next($i$)
        prev(next($h_1$)) ← $h_1$
        next($i$) ← $h_2$
        prev($h_2$) ← $i$
        $i$ ← $h_1$
        do
            face($i$) ← $f_1$
            $i$ ← next($i$)
        until target($i$) = $u$
        delete the face $f$
    }
```

In a similar fashion one can realize the inverse operation join-face(h) that removes the edge (represented by the halfedge) h, thereby joining the faces face(h) and face(twin(h)).

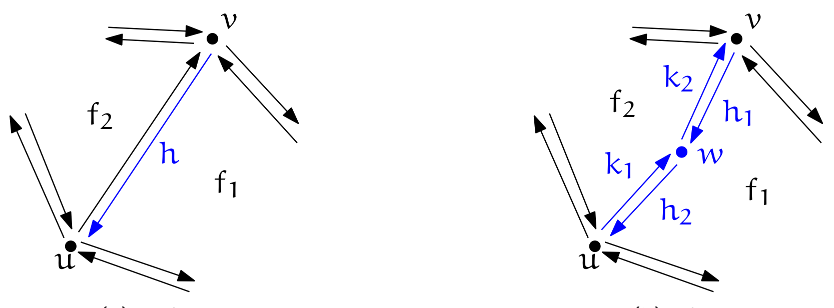
It is easy to see that every connected plane graph on at least two vertices can be constructed using the operations add-vertex-at and split-face, starting from an embedding of K_2 (two vertices connected by an edge).

Exercise 5.7 *Give pseudocode for the operation join-face*(h)*. Also specify preconditions, if needed.*

Exercise 5.8 *Give pseudocode for the operation split-edge*(h)*, that splits the edge (represented by the halfedge)* h *into two by a new vertex* w*, see Figure 5.5.*

5.2.2 Graphs with Unbounded Edges

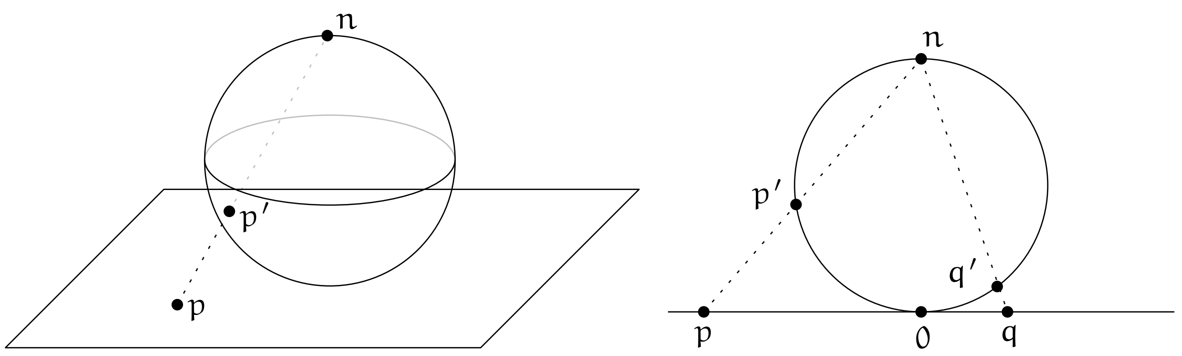
In some cases it is convenient to consider plane graphs, in which some edges are not mapped to a line segment but to an unbounded curve, such as a ray. This setting is not really much different from the one we studied before, except that one vertex is placed "at infinity". One way to think of it is in terms of *stereographic projection*: Imagine $\mathbb{R}^2$ being the x/y-plane in $\mathbb{R}^3$ and place a unit sphere S such that its southpole touches the origin. We obtain a bijective continuous mapping between $\mathbb{R}^2$ and $S \setminus \{n\}$, where n



(a) before (b) after

Figure 5.5: *Split an edge by a new vertex.*

is the northpole of S, as follows: A point $p \in \mathbb{R}^2$ is mapped to the point p' that is the intersection of the line through p and n with S, see Figure 5.6. The further away a point



(a) Three-dimensional view. (b) Cross-section view.

Figure 5.6: *Stereographic projection.*

in $\mathbb{R}^2$ is from the origin, the closer its image on S gets to n. But there is no way to reach n except in the limit. Therefore, we can imagine drawing the graph on S instead of in $\mathbb{R}^2$ and putting the "infinite vertex" at n.

All this is just for the sake of a proper geometric interpretation. As far as a DCEL representation of such a graph is concerned, there is no need to consider spheres or, in fact, anything beyond what we have discussed before. The only difference to the case with all finite edges is that there is this special infinite vertex, which does not have any point/coordinates associated to it. But other than that, the infinite vertex is treated in exactly the same way as the finite vertices: it has in– and outgoing halfedges along which the unbounded faces can be traversed (Figure 5.7).

5.2.3 Remarks

It is actually not so easy to point exactly to where the DCEL data structure originates from. Often Muller and Preparata [7] are credited, but while they use the term DCEL,

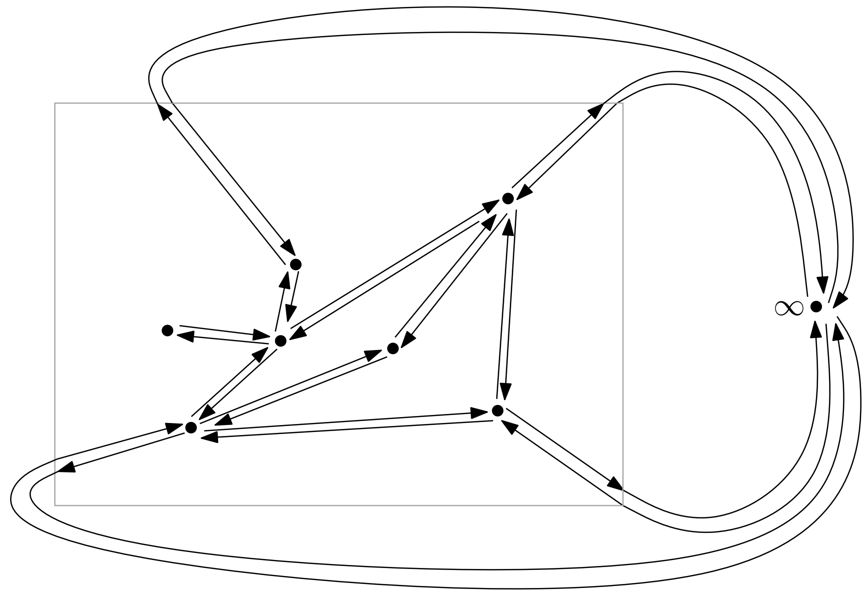


Figure 5.7: *A DCEL with unbounded edges. Usually, we will not show the infinite vertex and draw all edges as straight-line segments. This yields a geometric drawing, like the one within the gray box.*

the data structure they describe is different from what we discussed above and from what people usually consider a DCEL nowadays. Overall, there is a large number of variants of this data structure, which appear under the names *winged edge* data structure [1], *halfedge* data structure [9], or *quad-edge* data structure [5]. Kettner [6] provides a comparison of all these and some additional references.

Questions

17. *What are planar/plane graphs and straight-line embeddings?* Give the definitions and explain the difference between planar and plane.

18. *How many edges can a planar graph have? What is the average vertex degree in a planar graph?* Explain Euler's formula and derive your answers from it (see Exercise 5.2 and 5.3).

19. *How can plane graphs be represented on a computer?* Explain the DCEL data structure and how to work with it.

References

[1] Bruce G. Baumgart, A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.*, vol. 44, pp. 589–596, AFIPS Press, Alrington, Va., 1975, URL `http://dx.doi.org/10.1145/1499949.1500071`.

[2] John Adrian Bondy and U. S. R. Murty, *Graph Theory*, vol. 244 of *Graduate texts in Mathematics*. Springer-Verlag, New York, 2008, URL `http://dx.doi.org/10.1007/978-1-84628-970-5`.

[3] Reinhard Diestel, *Graph Theory*. Springer-Verlag, Heidelberg, 4th edn., 2010.

[4] István Fáry, On straight lines representation of planar graphs. *Acta Sci. Math. Szeged*, **11**, (1948), 229–233.

[5] Leonidas J. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, **4**, 2, (1985), 74–123, URL `http://dx.doi.org/10.1145/282918.282923`.

[6] Lutz Kettner, *Software design in computational geometry and contour-edge based polyhedron visualization*. Ph.D. thesis, ETH Zürich, Zürich, Switzerland, 1999, URL `http://dx.doi.org/10.3929/ethz-a-003861002`.

[7] David E. Muller and Franco P. Preparata, Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, **7**, (1978), 217–236, URL `http://dx.doi.org/10.1016/0304-3975(78)90051-8`.

[8] Klaus Wagner, Bemerkungen zum Vierfarbenproblem. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, **46**, (1936), 26–32.

[9] Kevin Weiler, Edge-based data structures for solid modeling in a curved surface environment. *IEEE Comput. Graph. Appl.*, **5**, 1, (1985), 21–40, URL `http://dx.doi.org/10.1109/MCG.1985.276271`.

[10] Douglas B. West, *An Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ, 2nd edn., 2001.