Last update: October 23, 2024 Translated From: e-maxx.ru

Finding faces of a planar graph

Consider a graph G with n vertices and m edges, which can be drawn on a plane in such a way that two edges intersect only at a common vertex (if it exists). Such graphs are called **planar**. Now suppose that we are given a planar graph together with its straight-line embedding, which means that for each vertex v we have a corresponding point (x, y) and all edges are drawn as line segments between these points without intersection (such embedding always exists). These line segments split the plane into several regions, which are called faces. Exactly one of the faces is unbounded. This face is called **outer**, while the other faces are called **inner**.

In this article we will deal with finding both inner and outer faces of a planar graph. We will assume that the graph is connected.

Some facts about planar graphs

In this section we present several facts about planar graphs without proof. Readers who are interested in proofs should refer to Graph Theory by R. Diestel (see also video lectures on planarity based on this book) or some other book.

Euler's theorem

Euler's theorem states that any correct embedding of a connected planar graph with n vertices, m edges and f faces satisfies:

$$n - m + f = 2$$

And more generally, every planar graph with k connected components satisfies:

$$n - m + f = 1 + k$$

Number of edges of a planar graph.

If $n \geq 3$ then the maximum number of edges of a planar graph with n vertices is $3n - 6$. This number is achieved by any connected planar graph where each face is bounded by a triangle. In terms of complexity this fact means that $m = O(n)$ for any planar graph.

Number of faces of a planar graph.

As a direct consequence of the above fact, if $n \geq 3$ then the maximum number of faces of a planar graph with n vertices is $2n - 4$.

Minimum vertex degree in a planar graph.

Every planar graph has a vertex of degree 5 or less.

The algorithm

Firstly, sort the adjacent edges for each vertex by polar angle. Now let's traverse the graph in the following way. Suppose that we entered vertex u through the edge (v, u) and (u, w) is the next edge after (v, u) in the sorted adjacency list of u. Then the next vertex will be w. It turns out that if we start this traversal at some edge (v, u), we will traverse exactly one of the faces adjacent to (v, u), the exact face depending on whether our first step is from u to v or from v to u.

Now the algorithm is quite obvious. We must iterate over all edges of the graph and start the traversal for each edge that wasn't visited by one of the previous traversals. This way we will find each face exactly once, and each edge will be traversed twice (once in each direction).

Finding the next edge

During the traversal we have to find the next edge in counter-clockwise order. The most obvious way to find the next edge is binary search by angle. However, given the counter-clockwise order of adjacent edges for each vertex, we can precompute the next edges and store them in a hash table. If the edges are already sorted by angle, the complexity of finding all faces in this case becomes linear.

Finding the outer face

It's not hard to see that the algorithm traverses each inner face in a clockwise order and the outer face in the counter-clockwise order, so the outer face can be found by checking the order of each face.

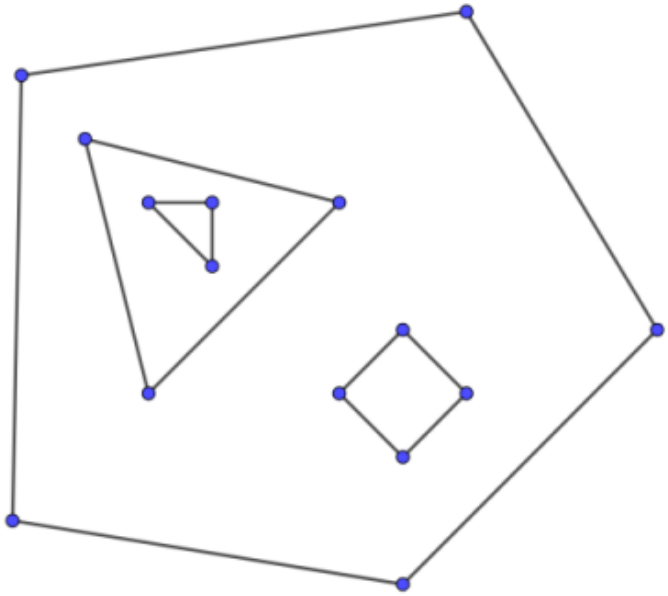
Complexity

It's quite clear that the complexity of the algorithm is $O(m \log m)$ because of sorting, and since $m = O(n)$, it's actually $O(n \log n)$. As mentioned before, without sorting the complexity becomes $O(n)$.

What if the graph isn't connected?

At the first glance it may seem that finding faces of a disconnected graph is not much harder because we can run the same algorithm for each connected component. However, the components may be drawn in a

nested way, forming **holes** (see the image below). In this case the inner face of some component becomes the outer face of some other components and has a complex disconnected border. Dealing with such cases is quite hard, one possible approach is to identify nested components with point location algorithms.



Implementation

The following implementation returns a vector of vertices for each face, outer face goes first. Inner faces are returned in counter-clockwise orders and the outer face is returned in clockwise order.

For simplicity we find the next edge by doing binary search by angle.

```
struct Point {
    int64_t x, y;

    Point(int64_t x_, int64_t y_): x(x_), y(y_) {}

    Point operator - (const Point & p) const {
        return Point(x - p.x, y - p.y);
    }

    int64_t cross (const Point & p) const {
        return x * p.y - y * p.x;
    }

    int64_t cross (const Point & p, const Point & q) const {
        return (p - *this).cross(q - *this);
    }

    int half () const {
        return int(y < 0 || (y == 0 && x < 0));
```

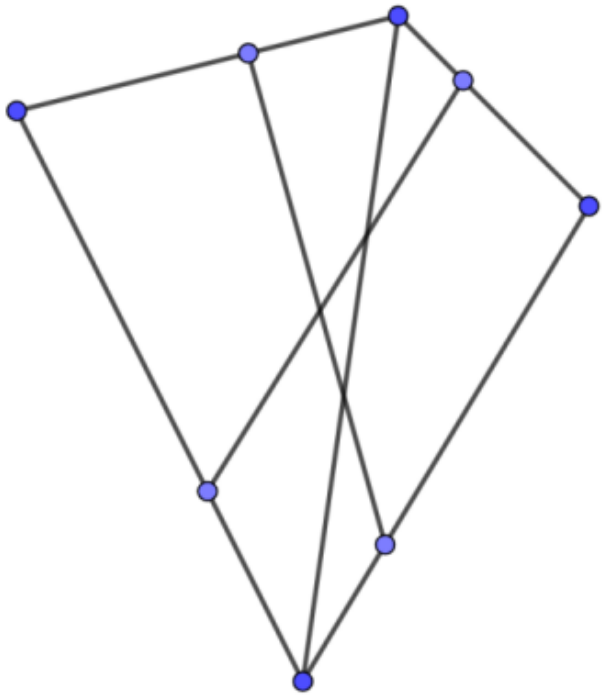
```
    }
};

std::vector<std::vector<size_t>> find_faces(std::vector<Point> vertices,
std::vector<std::vector<size_t>> adj) {
    size_t n = vertices.size();
    std::vector<std::vector<char>> used(n);
    for (size_t i = 0; i < n; i++) {
        used[i].resize(adj[i].size());
        used[i].assign(adj[i].size(), 0);
        auto compare = [&](size_t l, size_t r) {
            Point pl = vertices[l] - vertices[i];
            Point pr = vertices[r] - vertices[i];
            if (pl.half() != pr.half())
                return pl.half() < pr.half();
            return pl.cross(pr) > 0;
        };
        std::sort(adj[i].begin(), adj[i].end(), compare);
    }
    std::vector<std::vector<size_t>> faces;
    for (size_t i = 0; i < n; i++) {
        for (size_t edge_id = 0; edge_id < adj[i].size(); edge_id++) {
            if (used[i][edge_id]) {
                continue;
            }
            std::vector<size_t> face;
            size_t v = i;
            size_t e = edge_id;
            while (!used[v][e]) {
                used[v][e] = true;
                face.push_back(v);
                size_t u = adj[v][e];
                size_t e1 = std::lower_bound(adj[u].begin(), adj[u].end(), v, [&](size_t
l, size_t r) {
                    Point pl = vertices[l] - vertices[u];
                    Point pr = vertices[r] - vertices[u];
                    if (pl.half() != pr.half())
                        return pl.half() < pr.half();
                    return pl.cross(pr) > 0;
                }) - adj[u].begin() + 1;
                if (e1 == adj[u].size()) {
                    e1 = 0;
                }
                v = u;
                e = e1;
            }
            std::reverse(face.begin(), face.end());
            int sign = 0;
            for (size_t j = 0; j < face.size(); j++) {
                size_t j1 = (j + 1) % face.size();
                size_t j2 = (j + 2) % face.size();
                int64_t val = vertices[face[j]].cross(vertices[face[j1]],
vertices[face[j2]]);
                if (val > 0) {
                    sign = 1;
                    break;
                } else if (val < 0) {
                    sign = -1;
```

```
                    break;
                }
            }
            if (sign <= 0) {
                faces.insert(faces.begin(), face);
            } else {
                faces.emplace_back(face);
            }
        }
    }
    return faces;
}
```

Building planar graph from line segments

Sometimes you are not given a graph explicitly, but rather as a set of line segments on a plane, and the actual graph is formed by intersecting those segments, as shown in the picture below. In this case you have to build the graph manually. The easiest way to do so is as follows. Fix a segment and intersect it with all other segments. Then sort all intersection points together with the two endpoints of the segment lexicographically and add them to the graph as vertices. Also link each two adjacent vertices in lexicographical order by an edge. After doing this procedure for all edges we will obtain the graph. Of course, we should ensure that two equal intersection points will always correspond to the same vertex. The easiest way to do this is to store the points in a map by their coordinates, regarding points whose coordinates differ by a small number (say, less than 10^{-9}) as equal. This algorithm works in $O(n^2 \log n)$.



Implementation

```
using dbl = long double;

const dbl eps = 1e-9;

struct Point {
    dbl x, y;

    Point(){}
    Point(dbl x_, dbl y_): x(x_), y(y_) {}

    Point operator * (dbl d) const {
        return Point(x * d, y * d);
    }

    Point operator + (const Point & p) const {
        return Point(x + p.x, y + p.y);
    }

    Point operator - (const Point & p) const {
        return Point(x - p.x, y - p.y);
    }

    dbl cross (const Point & p) const {
        return x * p.y - y * p.x;
    }

    dbl cross (const Point & p, const Point & q) const {
        return (p - *this).cross(q - *this);
    }

    dbl dot (const Point & p) const {
        return x * p.x + y * p.y;
    }

    dbl dot (const Point & p, const Point & q) const {
        return (p - *this).dot(q - *this);
    }

    bool operator < (const Point & p) const {
        if (fabs(x - p.x) < eps) {
            if (fabs(y - p.y) < eps) {
                return false;
            } else {
                return y < p.y;
            }
        } else {
            return x < p.x;
        }
    }

    bool operator == (const Point & p) const {
        return fabs(x - p.x) < eps && fabs(y - p.y) < eps;
    }

    bool operator >= (const Point & p) const {
```

```
        return !(*this < p);
    }
};

struct Line{
    Point p[2];

    Line(Point l, Point r){p[0] = l; p[1] = r;}
    Point& operator [](const int & i){return p[i];}
    const Point& operator[](const int & i)const{return p[i];}
    Line(const Line & l){
        p[0] = l.p[0]; p[1] = l.p[1];
    }
    Point getOrth()const{
        return Point(p[1].y - p[0].y, p[0].x - p[1].x);
    }
    bool hasPointLine(const Point & t)const{
        return std::fabs(p[0].cross(p[1], t)) < eps;
    }
    bool hasPointSeg(const Point & t)const{
        return hasPointLine(t) && t.dot(p[0], p[1]) < eps;
    }
};

std::vector<Point> interLineLine(Line l1, Line l2){
    if(std::fabs(l1.getOrth().cross(l2.getOrth())) < eps){
        if(l1.hasPointLine(l2[0]))return {l1[0], l1[1]};
        else return {};
    }
    Point u = l2[1] - l2[0];
    Point v = l1[1] - l1[0];
    dbl s = u.cross(l2[0] - l1[0])/u.cross(v);
    return {Point(l1[0] + v * s)};
}

std::vector<Point> interSegSeg(Line l1, Line l2){
    if (l1[0] == l1[1]) {
        if (l2[0] == l2[1]) {
            if (l1[0] == l2[0])
                return {l1[0]};
            else
                return {};
        } else {
            if (l2.hasPointSeg(l1[0]))
                return {l1[0]};
            else
                return {};
        }
    }
    if (l2[0] == l2[1]) {
        if (l1.hasPointSeg(l2[0]))
            return {l2[0]};
        else
            return {};
    }
    auto li = interLineLine(l1, l2);
    if (li.empty())
        return li;
```

```
    if (li.size() == 2) {
        if (l1[0] >= l1[1])
            std::swap(l1[0], l1[1]);
        if (l2[0] >= l2[1])
            std::swap(l2[0], l2[1]);
        std::vector<Point> res(2);
        if (l1[0] < l2[0])
            res[0] = l2[0];
        else
            res[0] = l1[0];
        if (l1[1] < l2[1])
            res[1] = l1[1];
        else
            res[1] = l2[1];
        if (res[0] == res[1])
            res.pop_back();
        if (res.size() == 2u && res[1] < res[0])
            return {};
        else
            return res;
    }
    Point cand = li[0];
    if (l1.hasPointSeg(cand) && l2.hasPointSeg(cand))
        return {cand};
    else
        return {};
}

std::pair<std::vector<Point>, std::vector<std::vector<size_t>>>
build_graph(std::vector<Line> segments) {
    std::vector<Point> p;
    std::vector<std::vector<size_t>> adj;
    std::map<std::pair<int64_t, int64_t>, size_t> point_id;
    auto get_point_id = [&](Point pt) {
        auto repr = std::make_pair(
            int64_t(std::round(pt.x * 1000000000) + 1e-6),
            int64_t(std::round(pt.y * 1000000000) + 1e-6)
        );
        if (!point_id.count(repr)) {
            adj.emplace_back();
            size_t id = point_id.size();
            point_id[repr] = id;
            p.push_back(pt);
            return id;
        } else {
            return point_id[repr];
        }
    };
    for (size_t i = 0; i < segments.size(); i++) {
        std::vector<size_t> curr = {
            get_point_id(segments[i][0]),
            get_point_id(segments[i][1])
        };
        for (size_t j = 0; j < segments.size(); j++) {
            if (i == j)
                continue;
            auto inter = interSegSeg(segments[i], segments[j]);
            for (auto pt: inter) {
```

```
                curr.push_back(get_point_id(pt));
            }
        }
        std::sort(curr.begin(), curr.end(), [&](size_t l, size_t r) { return p[l] < p[r];
});
        curr.erase(std::unique(curr.begin(), curr.end()), curr.end());
        for (size_t j = 0; j + 1 < curr.size(); j++) {
            adj[curr[j]].push_back(curr[j + 1]);
            adj[curr[j + 1]].push_back(curr[j]);
        }
    }
    for (size_t i = 0; i < adj.size(); i++) {
        std::sort(adj[i].begin(), adj[i].end());
        // removing edges that were added multiple times
        adj[i].erase(std::unique(adj[i].begin(), adj[i].end()), adj[i].end());
    }
    return {p, adj};
}
```

Problems

- TIMUS 1664 Pipeline Transportation
- TIMUS 1681 Brother Bear's Garden

Contributors:

SYury (99.43%) Kostero (0.28%) adamant-pwn (0.28%)