Planarity testing of graphs

10.1 Introduction

We are interested in the following problem: given an undirected graph G, determine if G is planar ("can be drawn on a paper without having any two edges cross").

The algorithm developed in this lecture attempts to embed G in the plane. If it succeeds it will output the embedding (at this point we are deliberately vague about exactly what this means) and otherwise it will output "no." This is done by embedding one path at a time where the endpoints on the path are vertices that have already been embedded. Which path to embed, and how to embed it, is chosen in a greedy fashion. The difficulty is to show that the greedy choice used will always find an embedding, assuming one exists. The complexity of the algorithm is $O(|V|^2)$.

Hopcroft and Tarjan [1] present an algorithm that solves the problem in time $O(|V|)$ based on a previous algorithm [2, 3]. The algorithm presented here uses the same basic ideas but is less complicated.

10.2 Some fragments of graph theory

Let $G = (V, E)$ be a planar graph that has been embedded in the plane. The graph divides the plane into regions called *facets*. We let F be the set of facets. Note that the infinite region surrounding the graph is also a facet. See Figure 10.1.

The following relation, known as **Euler's formula**, holds for any planar graph $G = (V, E)$ (except for $|V| = 0$) where c is the number of connected components in G:

$$|V| + |F| - |E| = 1 + c.$$

This can easily be proven by induction.

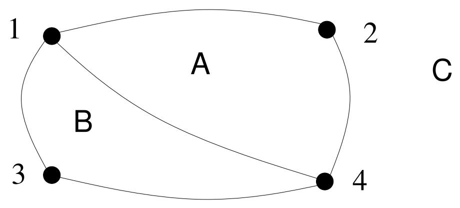

Figure 10.1: A graph embedded in the plane. $V = \{1,2,3,4\}$, $E = \big\{\{1,2\},\{1,3\},\{1,4\},\{2,4\},\{3,4\}\big\}$, and $F = \{\mathsf{A},\mathsf{B},\mathsf{C}\}$.

Lemma 1 *If $G = (V,E)$ is a planar graph, then $|E| \leq 3|V| - 6$.*

Proof: We use a simple counting argument. Assume that G has been embedded in the plane and that F is the set of facets. An edge $e \in E$ *touches* a facet $\mathsf{F} \in F$ if e is in F's cycle (i.e., the cycle that forms the boundary of F). We establish bounds on the size of $H = \{\,(e,\mathsf{F}) \mid e \text{ touches } \mathsf{F}\,\}$. Since every edge touches at most two facets we have $|H| \leq 2|E|$, and since every facet is touched by at least three edges we have $3|F| \leq |H|$ which implies

$$3|F| \leq 2|E|.$$

Combining this with Euler's formula (and observing that G has at least one connected component) proves the lemma. ■

From Euler's formula and the above lemma it follows that a planar graph has $|E| = \Theta(|V|)$, and for any embedding $|F| = O(|V|)$.

An undirected graph is biconnected if it is connected and at least two vertices must be removed to make it disconnected. Using depth-first search one can find all biconnected components of a graph (this induces a partition of E) in time $O(|V| + |E|)$. The following lemma explains the importance of biconnected components in planarity testing.

Lemma 2 *Let G be an undirected graph. Then G is planar if and only if each biconnected component is planar.*

This is easy to prove simply by observing that each biconnected component can be embedded in the plane separately.

10.3 A high-level description of the algorithm

The main part of the algorithm is the algorithm Embed which attempts to embed a biconnected graph in the plane.

Algorithm Embed

1. If $|E| \geq 3|V| - 6$ then output "no" and halt.
2. Find a (simple) cycle in G and embed it in the plane.
3. For each connected component C_j of un-embedded edges, let V_j be the set of embedded vertices that are incident on edges in C_j, and let F_j be the set of possible facets for C_j (a facet F is possible for C_j if all vertices in V_j are on the bounding cycle of F).
4. If $|F_j| = 0$ for any j, then output "no" and halt.
5. If $|F_j| = 1$ for some C_j, then take any path in C_j between two vertices in V_j and embed it in the single facet in F_j. If there are components left then go to 3, otherwise output the embedding.
6. If all components have $|F_j| \geq 2$ then pick a component C_i and a path from it and embed the path in some facet in F_i. If there are components left then go to 3, otherwise output the embedding.

The assumption that Embed deals with a biconnected graph implies that $|V_j| \geq 2$ for each component C_j in step 3.

There are several issues remaining to take care of. First of all we need to specify how to choose a component and a facet in step 6. Then we need to prove that the algorithm does not make bad choices in this step, that is, it will find an embedding if one exists. Finally, we need to show that the complexity of the algorithm is $O(n^2)$. We end this section by answering the first of these questions (and here we deviate slightly from the lecture): in step 6 the component C_i, the path, and the facet in F_i can be chosen arbitrarily. In the following sections we will argue that if an embedding exists then the algorithm will find it, and show that by somewhat careful bookkeeping the algorithm has complexity $O(n^2)$.

Algorithm Embed assumes that the graph is biconnected. To handle the case when G is not biconnected we introduce some pre and post processing.

Algorithm Main

1. If $|E| > 3|V| - 6$ then output "no" and halt.
2. Divide G into biconnected components $G_j = (V_j, E_j)$.
3. Run algorithm Embed on each component G_j. If Embed returns "no" for any G_j then output "no" and halt.
4. Combine the embeddings returned by Embed to a single embedding and output it.

10.4 Step 6 of Embed is sound

In this section we show that the choices in step 6 of the algorithm Embed can be made arbitrarily. The following is easy to check.

Lemma 3 *If we run the algorithm on a biconnected graph G, then, at any stage, the part that has been embedded is a biconnected graph.*

Now we get to the main lemma.

Lemma 4 *Let G be a biconnected graph which is given to the algorithm Embed. Assume that the algorithm reaches step 6 at time T and that we choose to embed a path from a component C_i that can be embedded in several facets. If G can be embedded by putting C_i in facet R and facet B is another possible facet in F_i, then G can be embedded by putting C_i in B.*

Proof: Both R and B are bounded by simple cycles since the embedded part of G is biconnected. We divide the vertices on these cycles into three groups: red, blue, and purple. The red vertices are those that occur in R's cycle but not in B's, the blue those that occur in B's cycle but not in R's, and the purple vertices are those that appear in both cycles. See Figure 10.2 for an example.

Since C_i has both R and B as possible facets, V_i must be a subset of the purple vertices. The components C_j that are un-embedded at time T and for which V_j consists only of purple vertices are called *purple components*.

Assume that we are given an embedding $\mathcal{E}$ which is consistent with the partial embedding that the algorithm has produced at time T and which embeds C_i in R. Consider

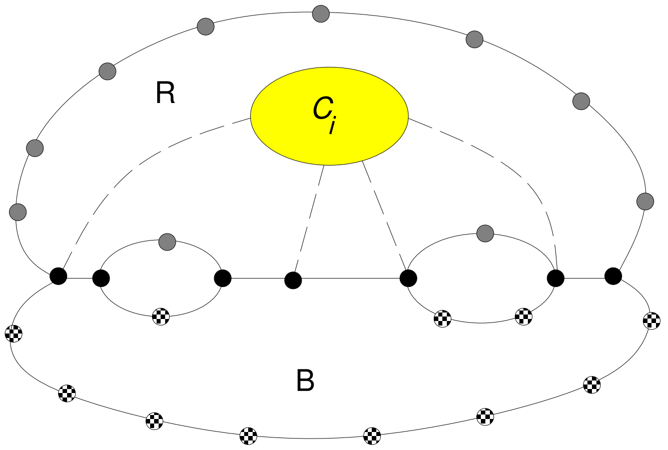

Figure 10.2: ● = red, ⊗ = blue, ● = purple. The component C_i has both R and B as possible facets. (Only the vertices and edges belonging to R and B are shown.)

the components that are un-embedded at time T and which $\mathcal{E}$ puts in either R or B. The purple ones are called *active* and the others are called *inactive*. We claim that we can get another embedding, where C_i is in B, by "reflecting" all active components, that is, flipping them from R to B and vice versa. Clearly this operation cannot cause a conflict between two active components or two inactive components. The only possibility is a conflict between an active component and an inactive component. We will show that no such conflict occurs in R, and by symmetry not in B.

Consider R's cycle. It is divided into *segments* in the following way. A purple segment is a maximal path of purple vertices. A red segment is a path of length ≥ 3 that has purple endpoints but all internal vertices are red (note that the endpoints of a segment are also endpoints of other segments).

Let C_l be an arbitrary active component in B and C_k an arbitrary inactive component in R. Assume that flipping C_l causes a conflict with C_k. Since C_k is inactive, V_k contains at least one red vertex that belongs to some segment S. If $V_k \subseteq S$, then there is no conflict with C_l, since C_l does not attach to any internal vertex of S. Thus, if there is a conflict, there must be some vertex (red or purple) in V_k that is not in S. However, in this case it is easy to see that at time T R is the only possible facet for C_k, and this contradicts the assumption that we have reached step 6. ■

The following theorem follows by straight-forward induction from Lemma 4.

Theorem 1 *Algorithm Main will find a planar embedding of G if one exists, and output "no" otherwise.*

10.5 Analysis of the algorithm

First it is necessary to specify how a (partial) embedding is represented. For each embedded vertex we have a list of its embedded neighbors and facets ordered clockwise. We call this a neighbor–facet list.

We also need some data structures for internal use in Embed. Rather than recomputing the components C_j and the sets V_j and F_j we will maintain a list of all un-embedded components, and for each component C_j a list F_j of its possible facets. For each facet F we maintain a list L_{F} of components that have F as a possible facet. The sets V_j will not be necessary to maintain.

Analysis of Main

Step 1 can certainly be done in time $O(|V|+|E|) = O(|V|^2)$ and with just about any reasonable representation of the input it can be done in time $O(|V|)$.

Step 2 can be done by depth-first search and hence in time $O(|V| + |E|)$ which is $O(|V|)$ by step 1.

A call to Embed costs $O(|E_j|^2)$ which will be shown below. Since $\sum_j |E_j| = |E| = O(|V|)$, the cost of step 3 is $O(|V|^2)$.

Finally we claim without proof that step 4 can be done in time $O(|V|)$.

Analysis of Embed

The input is a biconnected graph $G = (V, E)$. We need to show that embed can takes time $O(|E|^2)$.

Step 1 is easily done in $O(|V|+|E|)$, which is $O(|E|)$ since G is connected. As before, this can in fact be done in time $O(|V|)$.

Let $n = |V|$. After step 1 we have $|E| = O(n)$.

To find a cycle in step 2 only requires depth-first search which takes time $O(n)$. This is also sufficient time to construct the neighbor–facet lists for the vertices in the cycle, since each such list contains only two vertices and two facets. Again using depth-first search, we find all the connected components C_j and add them to a list *Comps*. For

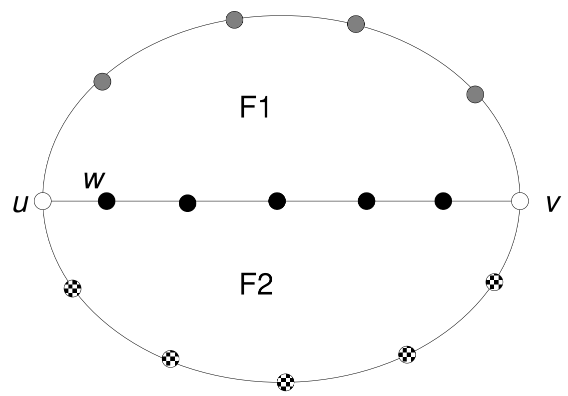

Figure 10.3: The facet F is split into F1 and F2 by embedding a new path. ● = type I, ⊗ = type II, ● = type III.

each C_j on $Comps$, let F_j be the list of the two facets A and B (inside and outside the cycle). Let L_{A} and L_{B} both be the list of all components. There are $O(n)$ lists of size $O(1)$ and $O(1)$ lists of size $O(n)$. The lists are easy to construct in time $O(n)$.

The steps 3 through 6 form a loop which is executed at most $|E| = O(n)$ times, since each iteration embeds at least one edge. We want to show that every step can be done in time $O(n)$.

Step 3 is actually unnecessary. We already have the lists that are needed, except for V_j, and these sets will not be needed explicitly.

Step 4 only involves checking if F_j is empty for any j, which is clearly doable in time $O(n)$ since there are $O(n)$ such lists. The same thing is true for checking if $|F_j| = 1$ for any j in step 5. If the algorithm checks if lists are empty or only contain one element after every update and maintains a list of components for which $|F_j| = 1$, then these steps only cost $O(1)$.

The only thing we need to worry about in step 5 and step 6 is the cost of finding the path in C_i and performing the necessary updates of our data structures. The path in C_i can be found by depth-first search in time linear in the size of C_j, which is certainly $O(n)$.

Embedding the path in a facet F splits it in two: F1 and F2 (see figure 10.3). Let u and v be the endpoints of the path, and let K be the cycle that is the boundary of the facet F, and let P be the path that is to be embedded in F.

Let w be u's neighbor on the new path. Update u's neighbor–facet list by replacing F by the sublist $\langle \mathsf{F1}, w, \mathsf{F2} \rangle$. Do similarly for v. Then, starting at u walk along K clockwise until v is reached. This can be done using the neighbor–facet lists. As we do this we replace each occurrence of F in an intermediate vertex's list by $\mathsf{F1}$. Also, mark each internal vertex on this u, v-path as "type I." Then walk from v to u clockwise along K replacing F by $\mathsf{F2}$ and mark internal vertices as "type II." The cost of this traversal is $O(deg(x))$ for each vertex x on K. Since there are $O(n)$ edges in the graph the combined cost for the traversal is $O(n)$.

Then mark all the internal vertices on P as "type III"[1] and construct their neighbor–facet lists (which is easy since they have two neighbors and $\mathsf{F1}$ and $\mathsf{F2}$ as their only facets). This clearly costs $O(n)$.

Finally, $Comps$ must be updated, the lists F_j must be modified for all components C_j that have F as a possible facet, and the lists L_{F1} and L_{F2} must be constructed. This is accomplished as follows. Make two empty lists L_{F1} and L_{F2}. Remove the component C_i, from which P was chosen, from $Comps$. C_i has been split into components C_{i_k} by the embedding of P. These components can be found by a depth-first search in C_i. Each new component is added to $Comps$. Note that each C_{i_k} has at least one type III vertex, so the only facets that could be possible for a new component is $\mathsf{F1}$ or $\mathsf{F2}$. Do a depth-first search of each component C_{i_k} to find V_{i_k}. By looking at the type labels on the vertices in V_{i_k} the lists F_{i_k} can be constructed as follows.

- If V_{i_k} contains both a type I vertex and a type II vertex, then F_{i_k} is set to an empty list (in this case we could halt the algorithm since it is impossible to embed C_{i_k}).
- If V_{i_k} contains a type I vertex but no type II vertex, then set F_{i_k} to $\langle \mathsf{F1} \rangle$ and add C_{i_k} to L_{F1}.
- If V_{i_k} contains a type II vertex but no type I vertex, then set F_{i_k} to $\langle \mathsf{F2} \rangle$ and add C_{i_k} to L_{F2}
- If V_{i_k} contains neither a type I vertex nor a type II vertex, then set F_{i_k} to $\langle \mathsf{F1}, \mathsf{F2} \rangle$ and add C_{i_k} to L_{F1} and L_{F2}.

It remains to handle all the "old" components C_j that had F as a possible facet. Note that C_j cannot attach to a type III vertex. For each component C_j on L_{F}, do a depth-first search to check if V_j contains any type I or type II vertex.

[1]The type III marking is not necessary, but it helps in the discussion.

- If V_j contains neither type, then remove F from F_j and add $\mathsf{F1}$ and $\mathsf{F2}$ (i.e., C_j attaches only to u and v), and add C_j to L_{F1} and L_{F2}.
- If V_j contains both type I and type II, then remove F from F_j.
- If V_j contains type I but not type II, then remove F from F_j, add $\mathsf{F1}$ to F_j, and add C_j to L_{F1}.
- If V_j contains type II but not type I, then remove F from F_j, add $\mathsf{F2}$ to F_j, and add C_j to L_{F2}.

Both for old and new components are searched in linear time and no edge appears in more than one component. This means that there are $O(n)$ insertions and deletions. We will not go into such detail as to show how the data structures should be implemented to allow each operation in $O(1)$ time. This is however possible using various cross-references between items in lists.

To summarize, the initial step of Embed costs $O(n)$, and each of the $O(n)$ iteration costs $O(n)$ showing that the complexity of Embed is $O(n^2)$.

References

[1] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. of the ACM*, 21:549–568, 1974.

[2] L. Auslander and S. V. Parter. On imbedding graphs in the plane. *J. Math. and Mech.*, 10:517–523, 1961.

[3] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in the plane. Graph and Combinatorics Conf., Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dep. of Math. Princeton U., May 16–18, 2 pp, 1963.