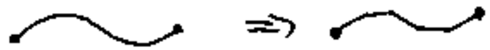
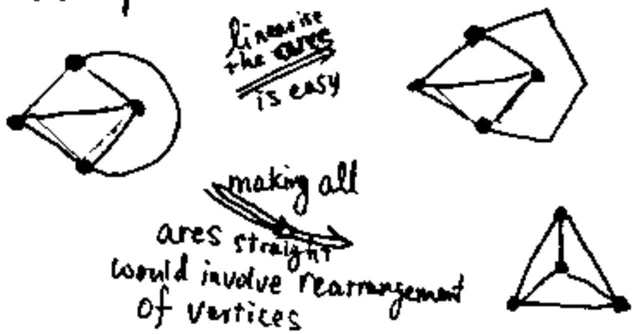
STRAIGHTENING AN EMBEDDING

We have seen that any [redacted] arc in a plane embedding could be "linearised" into finitely many consecutive line segments:



But it would be even more satisfactory if it ~~they~~ could be straightened into one segment. This task would be much harder since we have to move points around in some cases. For example:



The goal of this section is to give a positive answer for all planar graphs:

Theorem 21 (Fáry-Wagner)

Any planar graph could be embedded in $\mathbb{R}^2$ in a way that every edge is a straight segment. (arc)

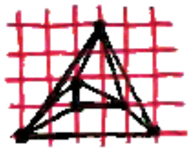
Even more surprisingly, we could even achieve a straight-embedding in $\mathbb{N}^2$ and in linear time!

Theorem 22 (de Fraysseix, Pach & Pollock)

Given a planar graph, one could compute in linear time an embedding of it s.t.

(1) all vertices lie on a $(2n-3) \times (n-1)$ grid (which is of course a subset of $\mathbb{N}^2$)

(2) all edges are straight line segments. (but they don't have to be horizontal/vertical, of course)



Its proof will be filling the pages to come, but the idea is actually simple: insert the vertices in an appropriate order, maintain the properties we need along the way, ~~and shift~~ shifting the vertices when necessary.

Before we do any actual work, some cleanups are in order. We observe that it suffices to <u>consider only maximal plane graphs</u>. The reason is:

- Given an arbitrary planar graph, we could find in linear time one of its embedding, by some variants of planarity testing algorithms.
- We then feed this embedding to the linear-time triangulation algorithm, which returns a maximal plane graph.
- We embed such maximal plane graph into the grid as advertised.
- Remove the excessive edges introduced by triangulation. And we are done.

Restricting our attention to only maximal plane graphs is advantageous since they are typically more "structured" than an arbitrary planar graph. This would save us the burden of considering every corner cases.

So ideally, our dream algorithm would incrementally build the desired embedding of a given maximal plane graph, where at each step we maintain the properties (1)(2). Here comes the problem: It might be really hard to make sure that the intermediate graphs are <u>maximal</u> planar. Well, in fact impossible: every time we add a vertex we must add exactly 3 edges to satisfy the maximality condition $|E| = 3|V| - 6$, so the final vertex always has degree 3. But clearly there's some maximal planar graph where every vertex has degree >3:

Hence the contradiction simply tells us: it's unrealistic to insist maximality every time we add a vertex!

The above discussion motivates the following definition of "almost maximal" plane graphs:

def. internally triangulated.

If all faces, except possibly the outerface, of a biconnected plane graph G are triangles, we call G internally triangulated. (By definition, any maximal plane graph is of course internally triangulated as well.)

The definition relaxes the restriction on outer face, thus giving us more freedom. Perhaps equally importantly, these graphs are still reasonably well-structured, which will aid our algorithm design.

e.g.  is internally triangulated but not maximal planar.

Now we are ready to address the very first challenge of "selecting an appropriate order to insert vertices".

def. Canonical ordering.

Let G be an internally triangulated plane graph. A canonical ordering of G is a vertex ordering $(v_1, \dots, v_n)$ s.t.

when inserting the vertices one by one in this way and from v_3 on:

(1) all intermediate graphs are internally triangulated.

(2) v_1v_2 always lies on the outer cycle.

(3) every vertex was in the outer face at the moment it was inserted.

This definition couldn't be more natural for our purpose. Point (1) essentially guarantees that the graph is well-structured all the way. Points (2)(3) ensures the graph "grows outwards" from a "base" v_1v_2.

Equivalently, one could state the definition in a "vertex removal", rather than "vertex insertion", fashion:

def' canonical ordering

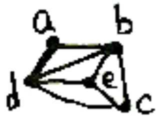
Let G be an internally triangulated plane graph. A vertex ordering $(v_1, \dots, v_n)$ is called a canonical ordering of G if, when removing vertices in order $v_n, \dots, v_3$,

(1) all intermediate graphs are internally triangulated.

(2) v_1v_2 always lies on the outer cycle.

(3) every removal happens on the outer cycle.

Exercise. See why the definitions are the same.

e.g. For graph 

,

(a, b, d, e, c), (b, a, d, e, c), (c, d, e, b, a) are all canonical orderings,

but $(a, b, \underline{e}, d, c)$, $(a, b, d, c, \underline{e})$ are not.

(1) violated (3) violated

Lemma 23

Every internally triangulated graph admits at least one canonical ordering. Moreover, we could compute one in linear time.

Proof. We propose a simple algorithm:

Algorithm Canonical Order

$C :=$ the outer cycle of G

$v_1v_2 :=$ an arbitrary edge on C

for $i = n \dots 3$ **do**

  choose $v_i \in C \setminus \{v_1, v_2\} : |N_G(v_i) \cap C| = 2$

  $G := G$ [redacted] $- v_i$

$\lfloor$ $C :=$ the new outer cycle of G

return $(v_1, v_2, \dots, v_n)$

This algorithm basically works in a vertex removal fashion as the second definition of canonical ordering suggests. Checking the correctness is straightforward:

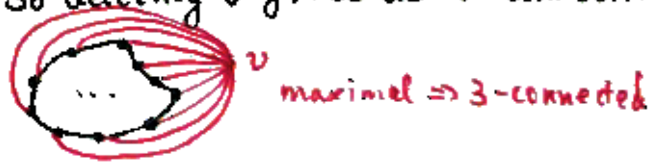
Suppose the algorithm works fine from n downto i, and we are ready to remove v_i. By the way that v_i is selected (assume for now that it exists), points (2)(3) are automatically satisfied. To see why (1) must hold as well, we draw a picture:

where the neighbours "in the middle" may or may not exist. Since G is internally triangulated, the neighbours of v_i must be connected in one path from u to w:

(Try to prove it formally!)

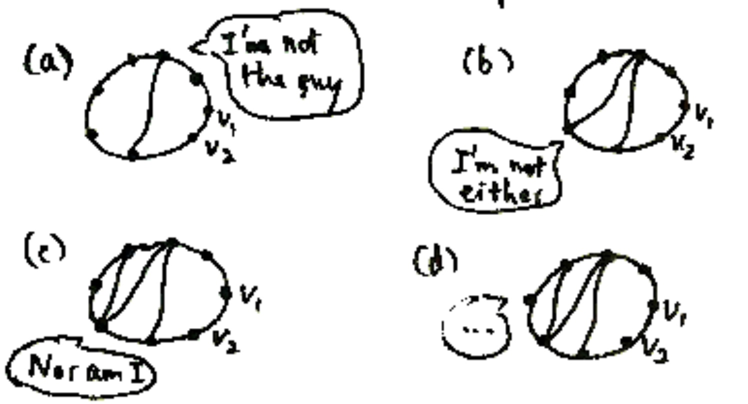
So after removing v_i, the path becomes part of the new outer cycle, and all the contents inside are still triangulated. Moreover, the new graph is biconnected ⟵ Any cut vertex in the new graph is also a cut vertex in G; but G is assumed biconnected.

(If you prefer a "high-level" argument, you could draw a virtual vertex v on the outer face and connect every possible edge to the outer cycle. The resulting graph is maximal planar, thus 3-connected. So deleting v gives us 2-connectivity.)



Since we have shown that removing v_i preserves both triangulation and biconnectivity, the new graph, $G - v_i$, is internally triangulated and thus (1) is true.

Now it remains to show that the advertised "$v_i \in C \setminus \{v_1, v_2\} : |N_G(v_i) \cap C| = 2$" indeed exists. The idea is simple:



The formalisation is left to the reader.

Finally, with standard trick, we could implement the algorithm in linear time:

Algorithm CanonicalOrder ; linear time

$$\text{outer}[v] := \begin{cases} \text{true} & v \in \text{outer cycle} \\ \text{false} & v \notin \text{outer cycle} \end{cases}$$

$\text{Count}[v] := |N_G(v) \cap (\text{outer cycle})|$ ← only for $v \in$ outer cycle

$v_1 v_2 :=$ an arbitrary edge on the outer cycle

$S := \{ v \in \text{outercycle} \setminus \{v_1, v_2\} : \text{Count}[v] = 2 \}$

for $i = n \ldots 3$ **do**

- take $v_i \in S$ arbitrarily
- let u and w be ~~the~~ two extremal neighbours (in the sense of circular order)

- remove v_i from G
- Count[u] --
- Count[w] --

} because v_i is gone

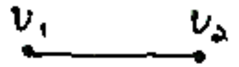
- **foreach** $v \in$ path $u \leadsto w$, excluding u & w **do**
  - mark v as "visited" in this loop
  - outer[v] := true
  - **foreach** $v' \in N_G(v)$ **do**
    - **if** outer[v'] and v' not visited **then**

prevent double counting

      - Count[v]++
      - Count[v']++
      - add/remove v & v' to/from S whenever the count reaches 2 / leaves 2.
- Clear the marks

∎

Now it's time to construct our straight-line embedding by inserting vertices in canonical order $(v_1, \dots, v_n)$. We draw $v_1 v_2$ as a straight-line first:



Then add v_3:

and v_4:

Now if v_5 connects to all of v_1, v_2, v_3 and v_4, we are in trouble. This example hints that we should carefully choose the location to put a vertex (say placing v_4 a bit to the left). A key idea of the algorithm in Theorem 22 is to put a new vertex above the others and "center" horizontally.

Proof of Theorem 22.

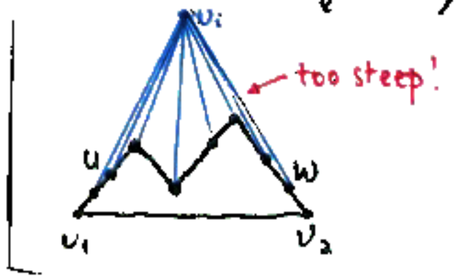
As usual, use C to denote the outer cycle. The algorithm shall maintain the invariant below:

<u>Invariant</u>

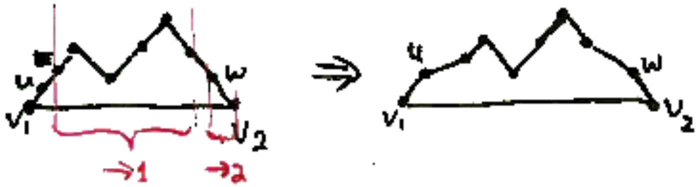
$v_1 v_2$ always lie on the x-axis;
all other edges in C has slope ± 1, and always go from left to right.

So the general shape of our embedding looks like:

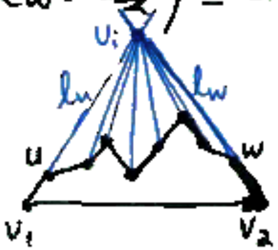
Every time we insert a new vertex v_i, we search for its leftmost vertex u and rightmost neighbour w, and roughly put v_i at the centre between u and w horizontally:



If we place v_i high enough, then clearly no crossing could occur. But then the slope would be too steep and the invariant breaks. So before actually placing v_i, we "stretch" the base shape a little. In particular, we shift the portition strictly between u and w to the right by unit distance; and the portion from w to v_2 to the right by distance 2:



Now we could safely place v_i at the intersection of l_u: $y=(x-x_u)+y_u$ and l_w: $y=-(x-x_w)+y_w$. Clearly the new outer cycle still has piecewise slope ± 1, so the invariant is preserved. Also, no crossing could occur



because the absolute slope of path $u \rightsquigarrow w$ is bounded by 1, but on the other hand the blue lines in the middle have ~~slope~~ absolute slope > 1.

Summarising the procedure formally:

Algorithm StraightLineEmbed

let $(v_1, \dots, v_n)$ be a canonical ordering of G

put v_1 at $(0,0)$ | bind $[v_1] := \{v_1\}$

put v_2 at $(0,2)$ | bind $[v_2] := \{v_2\}$

put v_3 at $(1,1)$ | bind $[v_3] := \{v_3\}$

for $i = 4 \dots n$ do

  bind $[v_i] := \{v_i\}$

  let u be the leftmost neighbour of v_i

  let w be the rightmost neighbour of v_i

  foreach $v \in$ outer cycle path $u \rightsquigarrow w$, excluding u & w do

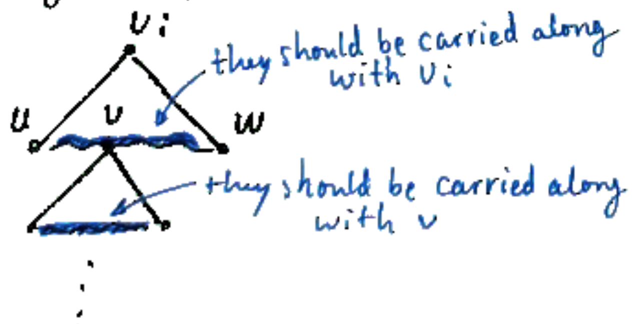
    append all vertices in bind$[v]$ to the set bind$[v_i]$; also, shift these vertices by unit distance

  foreach $v \in$ outer cycle path $w \rightsquigarrow v_2$ do

    shift all vertices in bind$[v]$ by distance 2

  put v_i at $\left(\frac{x_w+x_u+y_w-y_u}{2}, \frac{x_w-x_u+y_w+y_u}{2}\right)$

The most weird part in the description was perhaps the "bind". Indeed, it carries absolutely no geometry meaning. However, it has a very clear motivation. When we define the set bind[v_i], we are considering a future moment when v_i is shifted, and asking: which vertices should v_i carry along? Apparently, to avoid any accidental crossing, v_i must carry along the interior of $u \rightsquigarrow w$. But then, we have to ask recursively, if we ~~carry~~ shift those vertices, which further vertices should they carry along? And so on.



Staring at the algorithm long enough, one would realise that the computation of bind[v_i] is essentially expanding a recursion to obtain a set of vertices that has to move along with v_i.

But actually, bind[v_i] contains all the vertices we need to move:

Claim. Let $0 \le \delta_1 \le \delta_2 \le \cdots \le \delta_k$. Suppose the vertices on C are $u_1, u_2, \cdots, u_{k-1}, u_k$ from left to right. If we move bind[u_j] by distance δ_j (for all j), then the resulting drawing is still plane embedding.

One could easily show the claim by induction on the round of our algorithm. The reader is advised to do the proof — it surely helps appreciating the definition of bind!

By now the correctness of the algorithm is proved, i.e. it always produces a valid straight-line embedding. But some immediate observations are in order:

- Any $v, v' \in C$ have an even Manhattan distance
- Therefore $\frac{x_w + x_u + y_w - y_u}{2}$ and $\frac{x_w - x_u + y_w + y_u}{2}$ are integers, so the new vertex v_i is always on $\mathbb{N}^2$.
- So the algorithm produces in fact an embedding on square grid!
- v_2 is moved by 2 units each round, so it ends up at coordinate $2 + 2(n-3) = 2n-4$. v_1 always stay at the origin. So the embedding spans $2n-3$ horizontally. Consequently, it spans $1 + \frac{2n-4}{2} = n-1$ vertically.
- That is, the embedding lies on a $(2n-3) \times (n-1)$ grid! ■

Remark. To derive a linear time implementation, we have to make the shift operation more efficient. The natural idea is to organise the vertices in an abstract tree that corresponds to the "bind" relation. That is, u is a child of v if and only if $u \in bind[v] \setminus \{v\}$. Then, every time when we intend to shift all vertices in $bind[v]$, we refrain from that, but rather put a mark at node v indicating "the subtree is ordered to shift by ...". The technique is typically called "lazy operation", in the sense that we perform a shift virtually.

There are some details still. For example, we would use a binary-tree representation for the "binding tree" described above, because binary trees are easier to analyse and store. We won't present the entire picture here in the notes.