

Shortest paths with negative arc lengths.

Assume throughout there are no neg. length cycles (algorithms can report such cycles if exist).

So far we've only treated non-negative lengths.
Dijkstra's algorithm does not work.

In general graphs, the fastest known algorithm is Bellman-Ford.

Recall BF - n iterations.

On k 'th iteration compute shortest paths using at most k edges by relaxing all edges.

Running time $O(n \cdot m)$

We will see today how to do $O(n \log^2 n)$ for planar graphs.

Remarks:

- the fastest known is $O(n \frac{\log^2 n}{\log \log n})$
- these results generalize to bounded genus (later if we have time).

Price functions and reduced lengths:

Old idea (best known from Johnson's algo. for APSP in sparse graphs).

A price function is a function $\varphi: V \rightarrow \mathbb{R}$

the reduced length of arc uv w.r.t. φ is:

$$\text{len}_\varphi(uv) = \text{len}(uv) + \varphi(u) - \varphi(v)$$

lemma: for any s-to-t path P ,

$$\text{len}_\varphi(P) = \text{len}(P) + \varphi(s) - \varphi(t)$$

$$\text{len}_\varphi(P) = \sum_{uv \in P} \text{len}_\varphi(uv) = \sum_{uv \in P} \text{len}(uv) + \varphi(u) - \varphi(v)$$

$$\underset{\text{telescope}}{=} \text{len}(P) + \varphi(s) - \varphi(t)$$

□

Corollary: Shortest paths w.r.t. len , len_φ are the same.

φ is called a feasible price function if $\text{len}_\varphi(a) \geq 0$ for all arcs a .

Lemma: let $\delta(u)$ denote distances in G from some arbitrary node w . $\delta(u)$ is a feasible price function.

Pf: $\delta(v) \leq \delta(u) + \text{len}(uv)$ (^{shortest paths inequality})

so $\text{len}_g(uv) = \text{len}(uv) + \delta(u) - \delta(v) \geq 0$ □

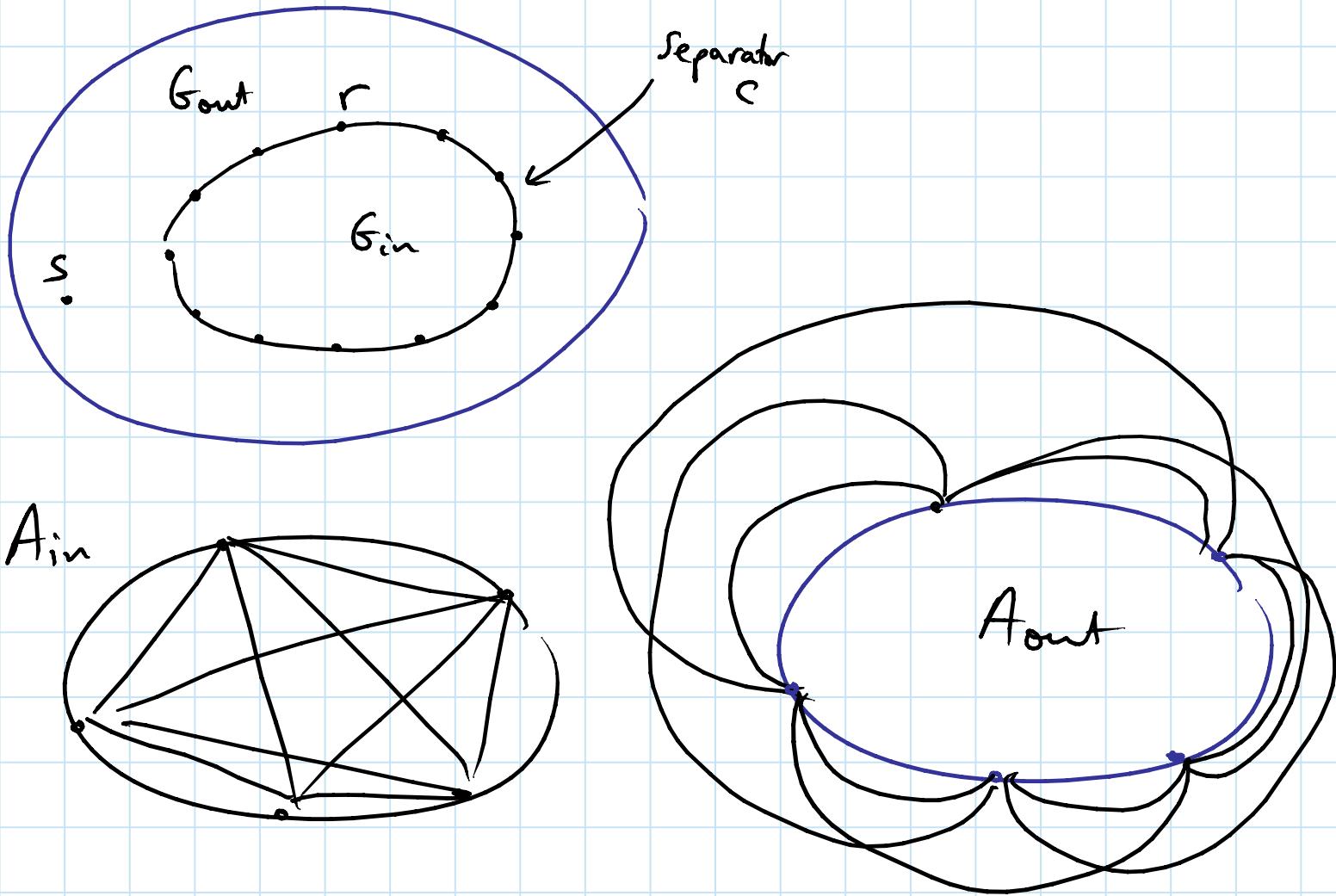
So, to compute shortest paths from u in G , suffices to compute shortest paths from any node v and then run Dijkstra.

So far we did not use planarity. Let's combine this approach with small separators.

$SP(G, s)$:

- find a cycle separator $C \rightarrow G_{in, out}$
- pick a node $r \in C$
- for $i \in \{in, out\}$ $\delta_i := SP(G_i, r)$

- ① - for $i \in \{in, out\}$ $A_i := APSP$ between nodes of C in G_i using MSSP (use δ_i as a feasible price function)
- ② - $B :=$ distances in G from r to nodes of C using BF on the complete graphs defined by A_{in} and A_{out}
called dense dist. graph
- ③ - for $i \in \{in, out\}$ $\delta'_i :=$ distances in G from r to all nodes of G_i using Dijkstra's algorithm initialized with B and price function δ_i
- $\delta' := \delta'_{in} \cup \delta'_{out}$ (distances from r in G)
- ④ - $\delta :=$ distances from s in G using Dijkstra's algorithm with price function δ'
- return δ



Analysis:

① uses 2 calls to MSSP $O(n \log n)$

② BF in graph with $O(\sqrt{n})$ nodes
and $O(n)$ edges $O(\sqrt{n} \cdot n) = O(n^{3/2})$

③, ④ SP with non-neg. lengths $O(n)$

$$T(n) \approx 2T\left(\frac{n}{2}\right) + O(n^{3/2}) = O(n^{3/2})$$

Bottleneck is BF. There are \sqrt{n} iterations.

At the beginning of k'th iteration we have an estimate $d(u)$ for the distance to each node u of C . We update the estimate by

$$\forall v \in C \quad d(v) := \min_{u \in C} \left\{ \begin{array}{l} d(u) + A_{in}(u, v) \\ d(u) + A_{out}(u, v) \end{array} \right\}$$

Equivalently, want to find all column minima

of the matrices $A'_i(u, v) = d(u) + A_i(u, v)$
 $(i \in \{\text{in, out}\})$

Naively this takes $O(n)$ time (the number of elements in A)

using the Monge property we can do it in $\tilde{O}(\sqrt{n})$

We say a matrix A is **Monge** if for $i < j$ and $k < l$ we have

$$A_{ik} + A_{jl} \leq A_{il} + A_{jk}$$

in fact, we will only use a weaker property.

a matrix A is **totally monotone** if for $i < j$, $k < l$

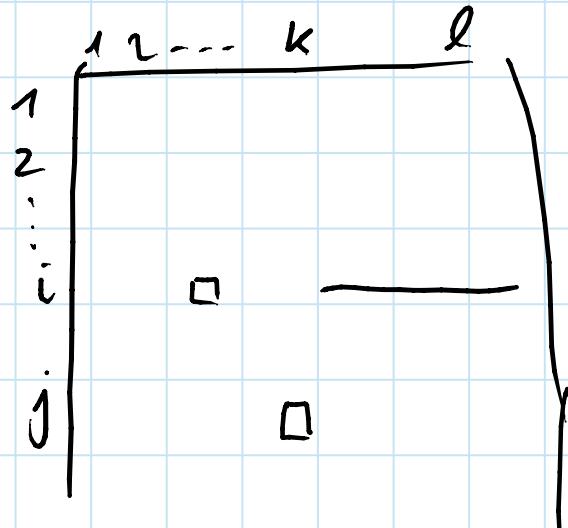
$$A_{ik} \geq A_{il} \Rightarrow A_{jk} \geq A_{jl}$$

clearly, if A is Monge, then both A and A^T are totally monotone.

We will show how to find row minima of a totally monotone $n \times n$ matrix in $O(n)$ time (Hence row and column minima of a Monge matrix in $O(n)$ time).

For simplicity will assume minima are unique

Claim: the sequence of column indices of row minima of a totally monotone matrix is monotonically increasing



Suppose k is minimum of row j
then $A_{jk} < A_{jl} \quad \forall l > k$,

so for $i < j$, $A_{ik} < A_{il} \quad \forall l > k$

so the column index of the min.
of row i is at most k .

Here is how we use this lemma in finding the row minima of a $n \times n$ totally monotone matrix.

First find the row minima of, say, just the even numbered rows. Then deduce in $O(n)$ time the minima of the even numbered rows.

So let's focus on $m \times n$ matrices with $m < n$.
call an element **dead** if it is not a row minimum

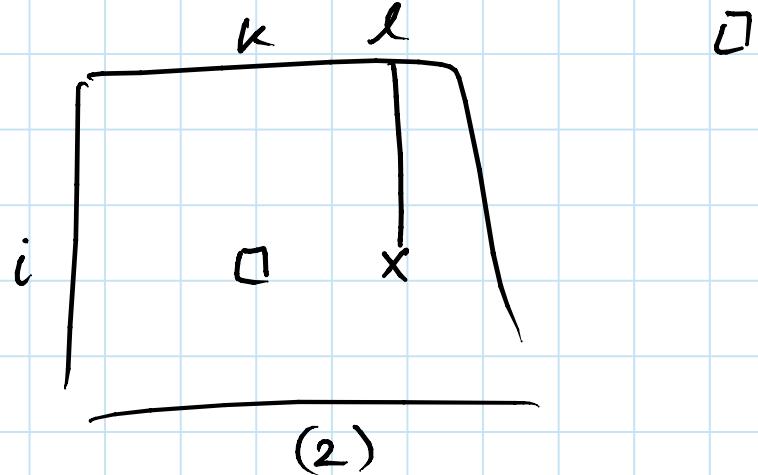
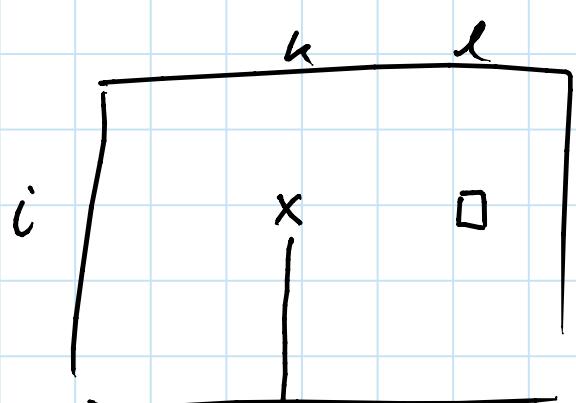
Lemma: Let A be a TM matrix. let $k < l$ be column indices.

(1) if $A_{ik} \geq A_{il}$ then $\forall j \geq i$ A_{jk} is dead

(2) if $A_{ik} < A_{il}$ then $\forall j \leq i$ A_{jl} is dead

Pf: (1) $A_{ik} \geq A_{il} \Rightarrow A_{jk} \geq A_{je}$

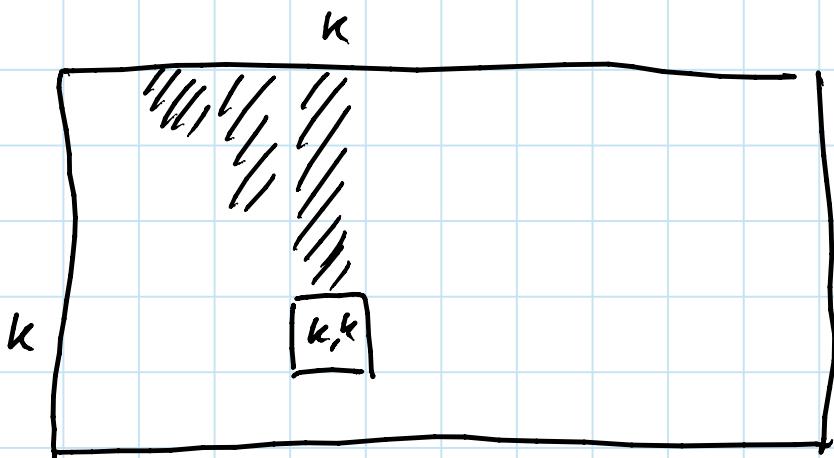
(2) suppose $A_{jk} > A_{jl} \Rightarrow A_{ik} \geq A_{il}$, contradiction.



We use this observation in a procedure that reduces a $m \times n$ TM matrix ($m \leq n$) into a $m \times m$ submatrix Q that include all columns with non-dead elements.

Define the index of Q to be the largest k s.t.

$$\forall j \leq k \quad \forall i < j \quad Q_{ij} \text{ is dead}$$



note : the index of every $m \times n$ matrix ($m \leq n$) is at least 1 and at most m

Suppose we know the index of Q is at least k .
Compare Q_{kk} with $Q_{k,k+1}$

- if $Q_{kk} < Q_{k,k+1}$ then by the previous lemma, all elements in column $k+1$ at or above k are dead.
if $k=m$ then column $m+1$ is entirely dead. otherwise the index of Q is at least $k+1$

- if $Q_{k,k} \geq Q_{k,k+1}$ then by the lemma, all elements of column k at or below k are dead
 so column k is entirely dead.

The reduction procedure is therefore:

Reduce(A):

$Q := A$

$k := 1$

while Q has more than m columns :

if $Q_{k,k} < Q_{k,k+1}$

if $k < n$ $k := k + 1$

else delete column $m+1$ of Q

else

delete column k of Q

if $k > 1$ $k := k - 1$

return Q

Analysis:

of deletions $\leq n - m$

of times k is decreased \leq # of deletions $\leq n - m$

of times k is increased $\leq m +$ # of times k decreased
 $\leq m + n - m = n$

of steps \leq # of increases + # of deletions $\leq 2n - m$
 $= O(n)$

it is easy to implement each operation in Reduce in constant time, so time for reducing an $m \times n$ matrix into a $m \times m$ one is $O(n)$

To complete the picture, here is the algo for computing all row minima of a totally monotone $m \times n$ matrix ($m \leq n$)

FindMin (A) :

1. if A has just one row, return the min column
2. $Q = \text{Reduce} (A)$
3. $Q' = \text{submatrix of } Q \text{ induced by even rows}$
4. FindMin (Q')
5. find the row minima of the odd rows of Q
6. return row minima of Q

step 2 takes $O(n)$ time

steps 3,5 take $O(m)$ time

let $T(m, n)$ denote the running time of FindMin on a $m \times n$ matrix

$$T(m, n) \leq c_1 n + c_2 m + T\left(\frac{m}{2}, m\right) = O(n)$$

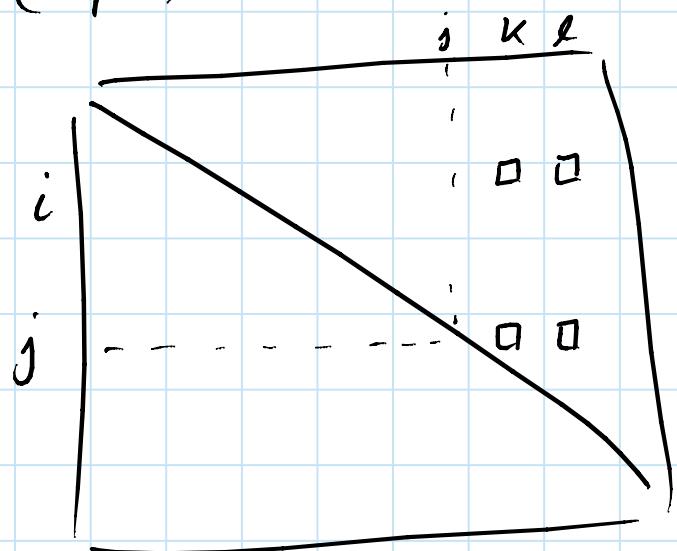
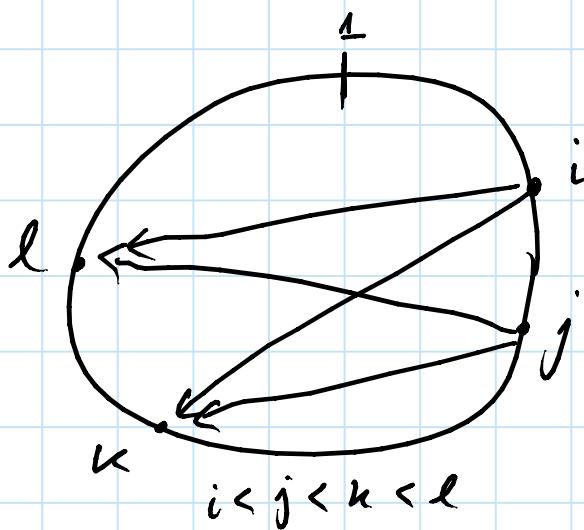
SMAWk - Aggarwal, Klawe, Moran, Shar, Wilber

Back to the BF step of the shortest paths algorithm. There are $\mathcal{O}(\sqrt{n})$ iterations, in each we compute

$$\forall v \in C \quad d(v) := \min_{u \in C} \left\{ \begin{array}{l} d(u) + A_{in}(u, v) \\ d(u) + A_{out}(u, v) \end{array} \right\}$$

that is, find all column minima of A' defined by

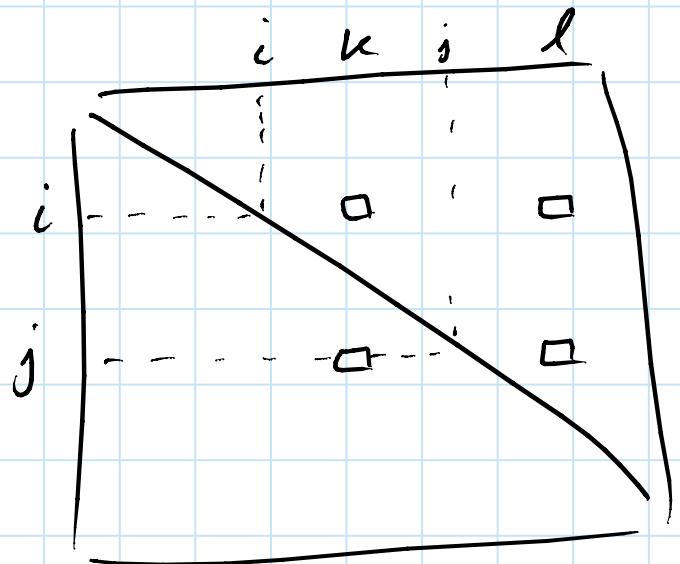
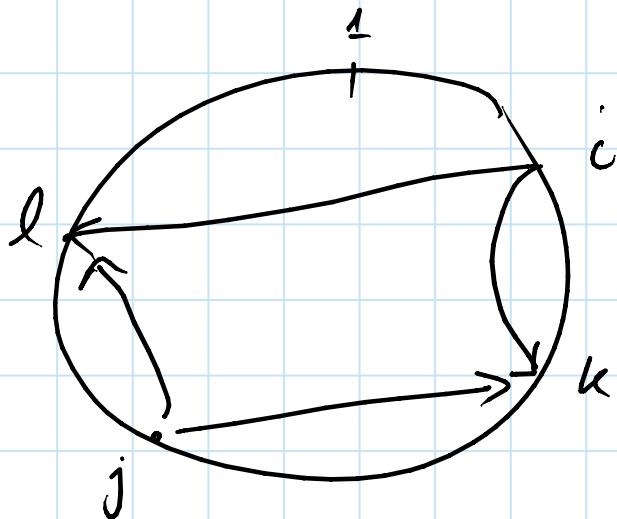
$$A'_{u,v} = d(u) + A_{in}(u, v)$$



$$d(i) + A_{in}(i, k) + d(j) + A_{in}(j, l) \geq$$

$$d(i) + A_{in}(i, l) + d(j) + A_{in}(j, k)$$

Monge property!

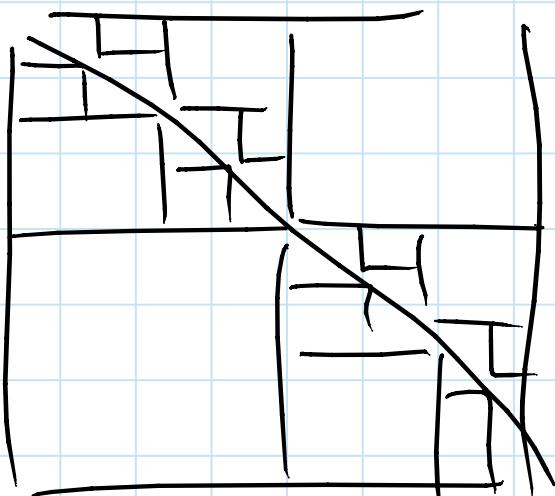


no Monge property ---

We saw one solution when discussed FR-Dijkstra:
matrix view:

\Rightarrow

BF step in
 $O(n \log n)$ time



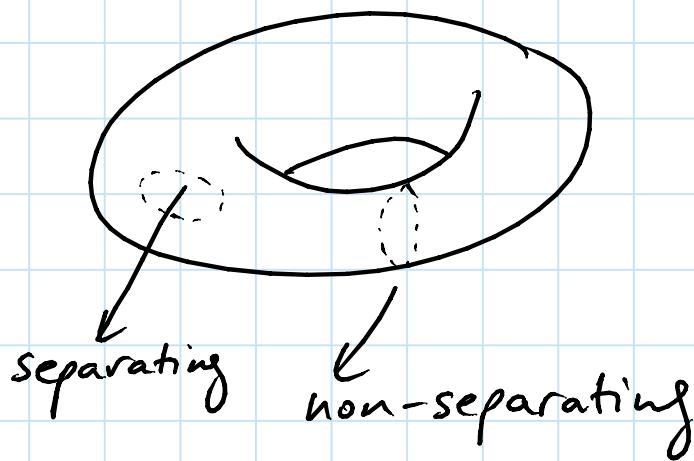
Another solution: adapt SMAWK to triangular matrices. Klawe, Kleitman - $O(n \alpha(n))$

\Rightarrow BF step in $O(n \alpha(n))$ time

fastest known result for SP uses this. time $O(n \frac{\log^2 n}{\log \log n})$

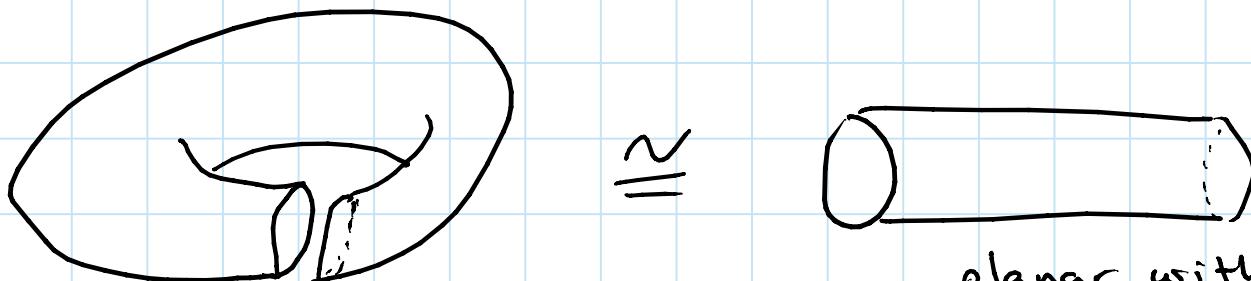
Extension to genus g :

Running time: $O(g^2 n \log^2 n)$ (roughly)



can find non-separating cycle C with $O(\sqrt{\frac{n}{g}} \log g)$ nodes in $O(g^2 n \log n)$ time

cut the graph open along C , duplicating C



planar with
2 boundary cycles

this reduces the genus by one.

Repeat until graph is planar. It has $2g$ boundary cycles. Each pair of boundary cycles corresponds to some non-separating cycle in the original graph

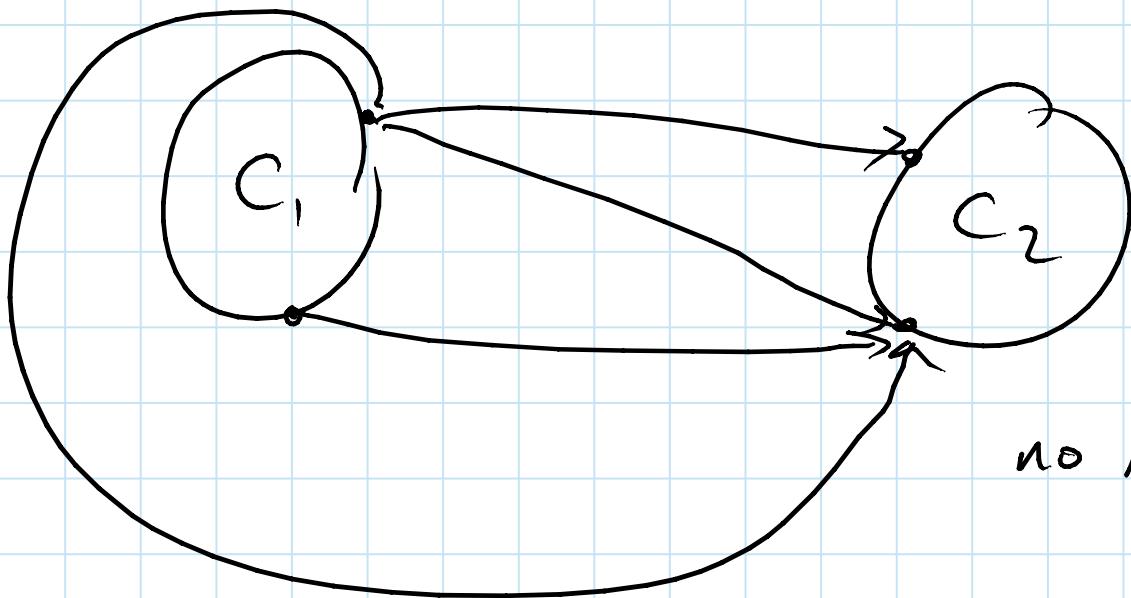
compute SP in the planar graph

use $2g$ calls to MSSP to compute all-pair distances between all nodes of all cycles, set the distances between the two copies of each boundary cycle node to zero.

Now run BF with these distances.

For each pair of boundary cycles (C_1, C_2) (possibly $C_1 = C_2$), relax the corresponding edges in the dense distance graphs together, using the Monge property.

- zero length edges (between two copies of a cycle) are few.
- we know how to relax edges between nodes of the same cycle.
- how about edges between two different cycles?



you will show in the problem set how to handle those

Running time:

$2g$ cycles, each with $O(\sqrt{\frac{n}{g}} \log g)$ nodes.

total # of nodes (# of iterations for BF) is
 $O(\sqrt{ng} \log g)$

number of pairs $O(g^2)$

relaxing all edges between a pair of different cycles
takes $O(\sqrt{\frac{n}{g}} \log g)$ time.

\Rightarrow time for a single iteration of BF:

$$O(g^2 \cdot \sqrt{\frac{n}{g}} \log g)$$

\Rightarrow total time for BF:

$$O(g^2 n \log^2 g)$$

time for SP in planar graph is $O(n \log^2 n)$

so overall time is bounded by $O(g^2 n \log^2 n)$.