Lecture 16

Graph Sketches: Dynamic Spanning Forest

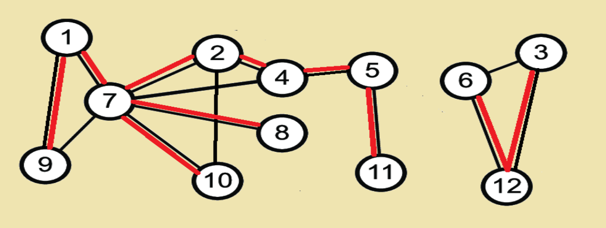
Course: Algorithms for Big Data

Instructor: Hossein Jowhari

Department of Computer Science and Statistics
Faculty of Mathematics
K. N. Toosi University of Technology

Spring 2021

Spanning forest



Given a graph $G = (V, E)$, a spanning forest of G is a forest $F = (V, E')$

- $E' \subseteq E$.
- Adding an edge $e \in E/E'$ to F does not change the number of connected components in F.

Fact: a graph can have multiple spanning forests.

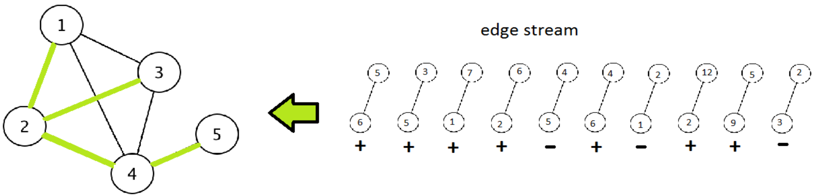
Spanning forest: applications

- Algorithm design: intermediate step in designing graph algorithm (connectivity and path finding)
- Graph analysis: spanning forest gives the connected components of a graph.
- Network design: Minimum number of links to keep the nodes connected.

Computing a spanning forest

Given a graph on n vertices and m edges, we can compute a spanning forest in time $O(m)$ and $O(m)$ space (BFS/DFS graph traversal).

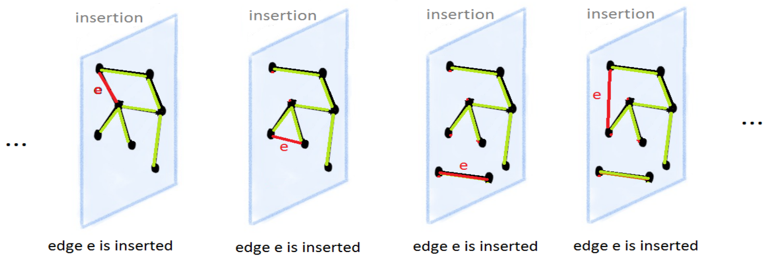
What if the graph is dynamic (edges are inserted and deleted) and we have to <u>maintain</u> a spanning forest?



Spanning forest: insert-only streams

When the stream is series of edge insertions, maintaining a spanning forest is easy.

If the new inserted edge e does not create a cycle it is added to the forest otherwise it is ignored.

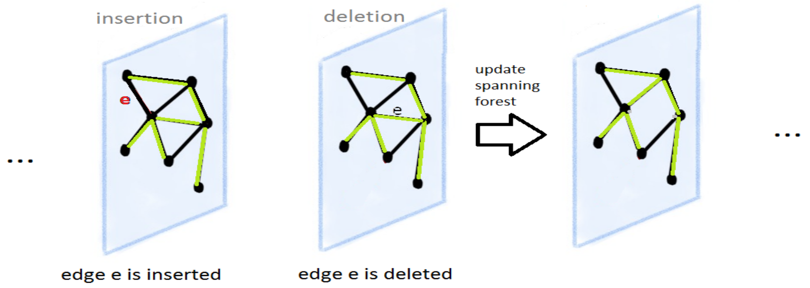


Space usage: $O(n \log n)$ bits

per-edge time: ?

Spanning forest: dynamic graphs

When the edges, in addition to being inserted, are deleted as well it is not clear how to maintain a spanning forest without storing all existing edges.



Dynamic spanning forest via ℓ_0 sampling

[Ahn, Guha, McGregor, 2012] There is a randomized algorithm for maintaining a spanning forest under insertion/deletion of edges that uses $O(n \log^3 n)$ bits of space. The algorithm uses ℓ_0 sampling as a subroutine.

ℓ_0 sampling: Given a stream of positive and negative updates on a vector $\boldsymbol{x} \in \mathbb{R}^n$, a ℓ_0 sampler is a randomized algorithm that returns a random non-zero coordinate $i \in [n]$ where the probability of returning each non-zero coordinates is

$$\frac{1}{\|\boldsymbol{x}\|_0} \pm \frac{1}{n^c}.$$

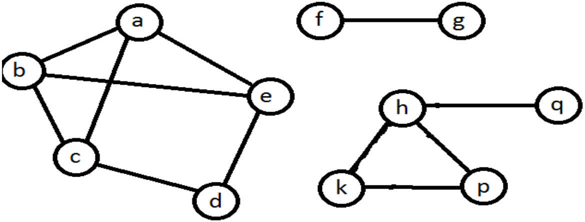
With probability at most δ, the algorithm might declare failure and return no sample. The algorithm uses $O(\log^2 n \log(\frac{1}{\delta}))$ bits of space.

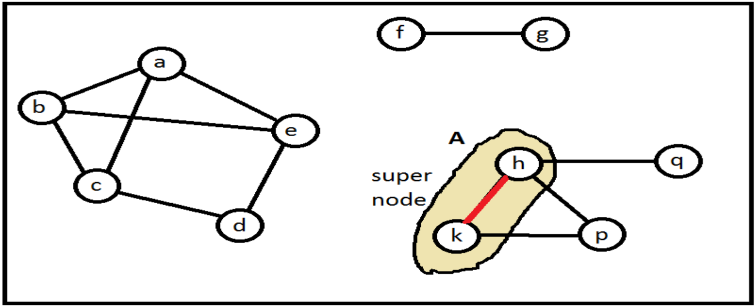
Computing the connected components

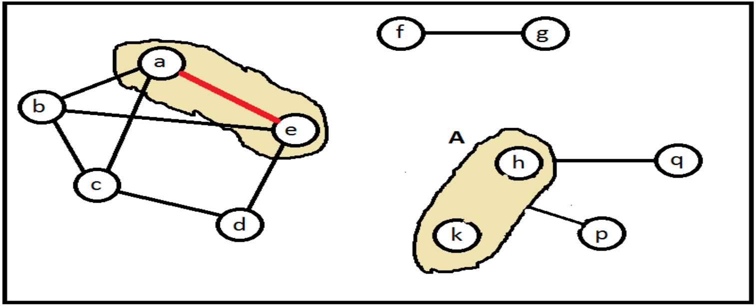
Lets consider a simpler problem: report the connected components after all edge-insertion/deletions are done.

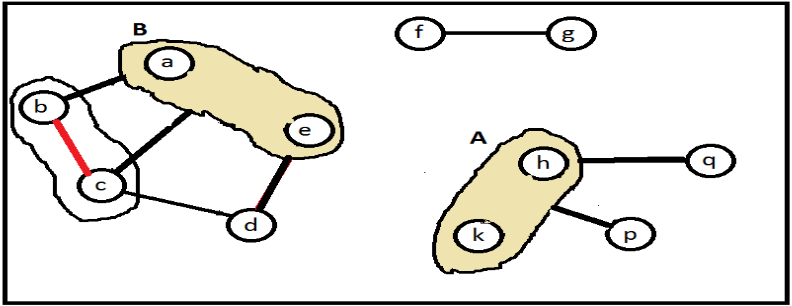
The algorithm is based on a simple strategy:

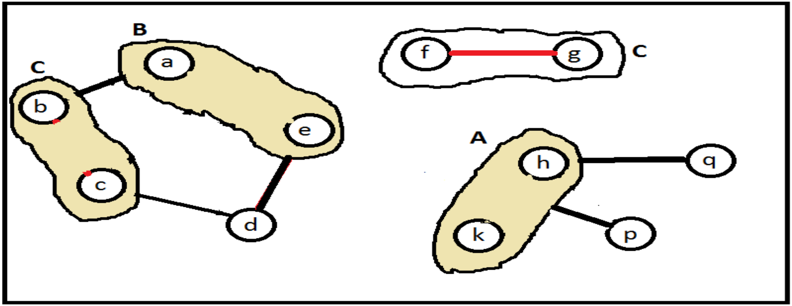
Each time pick a random edge and merge its two endpoints into a super-node. Continue this process until no edge remains. In the end, the isolated super-nodes represent the connected components.



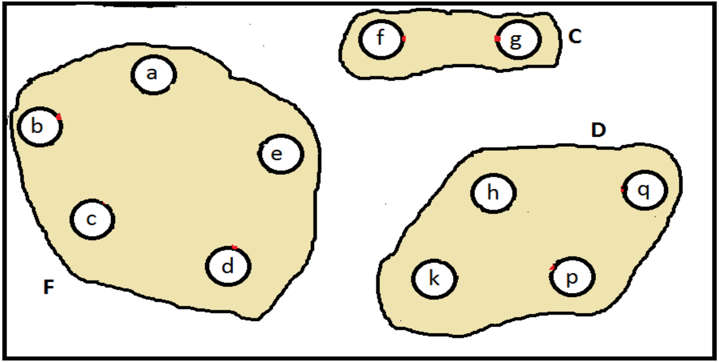

f
g
a
b
e
c
d
A
h
q
super
node
k
p


f
g
a
b
e
c
d
A
h
q
k
p

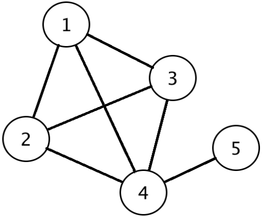

B
a
b
c
e
d
f
g
A
h
q
k
p


B
C
a
b
c
e
d
f
g
C
A
h
q
k
p

In the end, the connected components remain as isolated super-nodes.

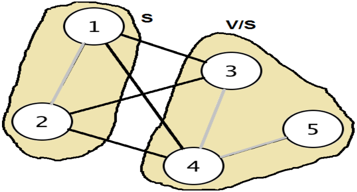


We can pick a random edge by ℓ_0 sampling the adjacency matrix A.



$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

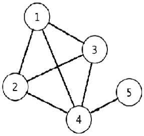
However, in addition to this, we want to be able to sample an edge from the cut $(S, V/S)$ when S is a super-node. How can we do this?



Suppose the vertices are labeled by numbers in $\{1, 2, \dots, n\}$.

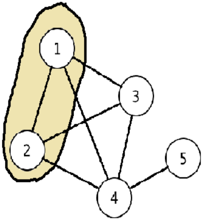
For each node $i \in V$, we define a vector $\boldsymbol{u}_i \in \{-1, 0, +1\}^{\binom{n}{2}}$ as follows.

- If the edge (i, j) exists and $i < j$ then we set the coordinate $\boldsymbol{u}_i(i, j) = +1$
- If the edge (i, j) exists and $i > j$ then we set the coordinate $\boldsymbol{u}_i(i, j) = -1$

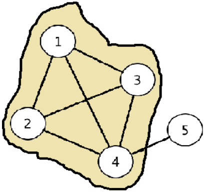


	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
u1	+1	+1	+1	0	0	0	0	0	0	0
u2	-1	0	0	0	+1	+1	0	0	0	0
u3	0	-1	0	0	-1	0	0	+1	0	0
u4	0	0	-1	0	0	-1	0	-1	0	+1
u5	0	0	0	0	0	0	0	0	0	-1
u1+u2+u3+u4	0	0	0	0	0	0	0	0	0	+1

The vector $\boldsymbol{u}_{i_1} + \ldots + \boldsymbol{u}_{i_r}$ corresponds to the super-node $S = \{u_{i_1}, \ldots, u_{i_r}\}$.



	(1,2)	(1,3)	(1,4)	(1,5)	(2,3)	(2,4)	(2,5)	(3,4)	(3,5)	(4,5)
u1	+1	+1	+1	0	0	0	0	0	0	0
u2	-1	0	0	0	+1	+1	0	0	0	0
u1+u2	0	+1	+1	0	+1	+1	0	0	0	0



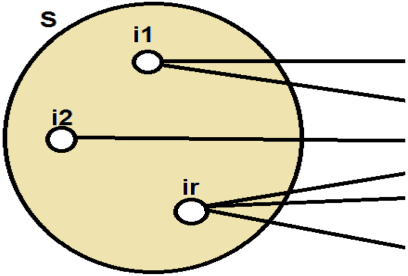
u1	+1	+1	+1	0	0	0	0	0	0	0
u2	-1	0	0	0	+1	+1	0	0	0	0
u3	0	-1	0	0	-1	0	0	+1	0	0
u4	0	0	-1	0	0	-1	0	-1	0	+1
u1+u2+u3+u4	0	0	0	0	0	0	0	0	0	+1

For each vector $\boldsymbol{u}_i$, we maintain an ℓ_0 sampling sketch $sk(\boldsymbol{u}_i)$.

$$sk(\boldsymbol{u}_1),\ sk(\boldsymbol{u}_2),\ \ldots\ ,\ sk(\boldsymbol{u}_n)$$

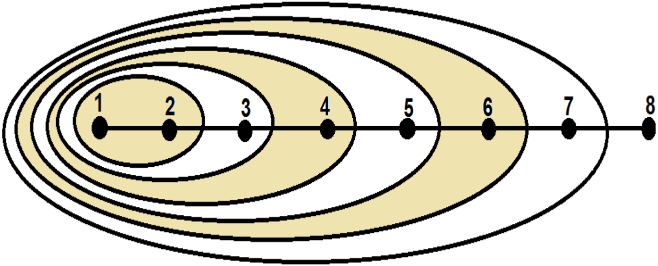
If we want to sample an edge from the cut $(S, V/S)$ where $S = \{i_1, i_2, \ldots, i_r\}$, we use the sketch

$$sk(\boldsymbol{u}_{i_1} + \boldsymbol{u}_{i_2} + \ldots + \boldsymbol{u}_{i_r})$$



There is one problem: if we contract the edges, one edge at a time, we may end up using the sketch $sk(\boldsymbol{u}_1)$ multiple times ($n-1$ times!)

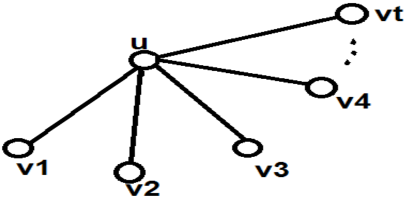
ℓ_0 sample $\boldsymbol{u}_1 : sk(\boldsymbol{u}_1)$
ℓ_0 sample $\boldsymbol{u}_1 + \boldsymbol{u}_2 : sk(\boldsymbol{u}_1 + \boldsymbol{u}_2)$
ℓ_0 sample $\boldsymbol{u}_1 + \boldsymbol{u}_2 + \boldsymbol{u}_3 : sk(\boldsymbol{u}_1 + \boldsymbol{u}_2 + \boldsymbol{u}_3)$
...
ℓ_0 sample $\boldsymbol{u}_1 + \boldsymbol{u}_2 + \ldots + \boldsymbol{u}_{n-1} : sk(\boldsymbol{u}_1 + \boldsymbol{u}_2 + \ldots + \boldsymbol{u}_{n-1})$



The ℓ_0 sample drawn from $\boldsymbol{u}_1 + \boldsymbol{u}_2$ will depend on the ℓ_0 sample drawn from $\boldsymbol{u}_1$. Dependency!!

We should not use the sketch $sk(\boldsymbol{u}_i)$ multiple times because it will cause dependency issues.

If we could query a sketch multiple times we could find all neighbors of a node by using only $O(\log^3 n)$ bits of space! This is impossible because one cannot compact $\Omega(n)$ bits of information in $\log^3 n$ bits of space.

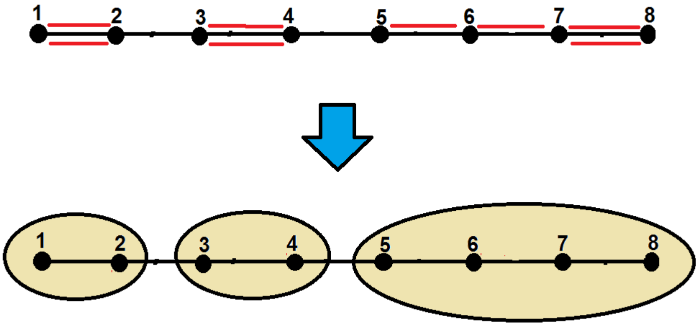


How to avoid using a sketch multiple times?

Lets assume there is no isolated vertex in the input graph $G = (V, E)$.

The algorithm works in multiple rounds. In the first round we do the following:

- For each vertex $i \in V$, we maintain an independent ℓ_0 sampling sketch $sk_1(\boldsymbol{u}_i)$.
- We ℓ_0 sample the vector $\boldsymbol{u}_i$ using the sketch $sk_1(\boldsymbol{u}_i)$. As result, for each vertex $i \in V$, we find a random neighbor of i.
- We find at least $\frac{n}{2}$ random edges in the first round.
- We contract the random edges and create the super-nodes.



In each round, number of nodes drops by a factor of $\frac{1}{2}$.

number of nodes in the first round = n
number of nodes in the second round $\leq \frac{n}{2}$

As result, the algorithm finishes in at most $\log n$ rounds.

In each round we use fresh ℓ_0 sampling sketches for all vertices.

Since there are at most $\log n$ rounds, for each vertex we need to maintain $\log n$ number of independent ℓ_0 sampling sketches.

In each round, we pick one of the sketches that are not used previously.

In total, we use $n \log n$ number of ℓ_0 sketches. Each sketch takes $O(\log^2 n)$ bits of space. Therefore the space complexity is $O(n \log^3 n)$ bits.