Spanning trees (Lecture 1)

MST: Classical Algos:

Kruskal		'56	$O(m\log m)$
Prim ('30.) (Jarnik ('30) / Dijkstra ('59? '57?))		'57	$O(m\log n)$ → $O(m+n\log n)$
Boruvka		'26	$O(m\log n)$

Modern:

- Yao '75 $O(m\log\log n)$
- Fredman Tarjan 84 $O(m\log^* n)$
- Gabow Galil Spencer Tarjan 85? $O(m\log\log^* n)$
- Karger (Karger Klein Tarjan) '94 [Randomized]
- Chazelle '97 $O(m\,\alpha(m,n))$ inverse ackermann fn.
-

all use blue rule

Basic Idea of MST algorithms: Given $G=(V,E)$, edge weights can compare in $O(1)$ time (don't use facts about edge wts.)

~~Cut rule~~:

m edges, n nodes, connected.

- can assume no parallel edges
- assume nodes are numbered $1..n$.
- assume all edge wts are distinct.

Cut rule: take a ~~the~~ cheapest edge across some cut. this is in an MST. (color it blue)

Cycle rule: take any cycle and drop the heaviest edge on the cycle (color it red).

Thm: Repeatedly apply cut and cycle rules in any order. get the MST.

Most of simpler algorithms (all the ones in lec#1) just use the cut rule.

These facts also extend to "matroids" [See HW 1] where we can define the notion of a cut and cycle analogously. But algos often use more of the graph.

(2)

<u>Algo #1</u>: Kruskal. Sort edges. Add next edge if it does not create cycle (if it decreases # of components) (if it crosses a cut in the current graph).

How to check if edge crosses a cut? Disjoint set union (aka Union find) data structure.

<u>Operations</u>: createset.

find (e) ← returns "root" of the set containing e

Union (r_1, r_2) ← unions the sets whose roots are r_1, r_2.

Using union-by-rank and path compression:–

any set of k find + (unions + create set) ← at most $O(n)$ operations takes time ~~$k\alpha(n)$ time~~.
(on a universe of size n) $O(n + k \log n)$

↓

$O(n + k\,\alpha(k,n))$

$A(m,n) = \begin{cases} n & \text{for } m=0 \\ A(m-1,1) & \text{for } n=0 \\ A(m-1, A(m,n-1)) \end{cases}$

~~At~~

m\n	0	1	2	3	4
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29		

$2n$

2^{4}

Various def^{n}s but ~~almost~~ all essentially this:

$A(m,n) = \begin{cases} 2n & \text{for } m=0 \\ 2 & \text{for } n=0. \\ A(m-1, A(m,n-1)) \end{cases}$

	1	2	3	4	5	
1	2	4	6	8	10	
2	2	4	8	16		2^n
3	2	4	2^{2^2}	$2^{2^{2^2}}$		tower.
4	2	4	$2^{2^{2^2}}$			

Runtime: $O(m \log m) + O(n + m\,\alpha(m,n)) = O(m \log m)$.
↑
sort

③

Prim's Algorithm: Pick a root vertex: Maintain some root component. C

Pick smallest ~~edge~~ wt edge out of root component and add it in, (Blue rule)

So want to maintain: for each vertex v not in C, the length of shortest edge from C to v. (if any). And quickly want to find min.

Priority queue: each element (node) has a key (wt of edge).

Bin H ary	Fib H	
$\log n$	$\boxed{O(1)}$	• insert(e, k)
$\log n$	$O(\log n)$	• delete ~~find~~ min (return element with smallest key, remove from pqueue)
$\log n$	$\boxed{O(1)}$	• decreasekey(e, k').
$\log n$	$O(\log n)$	• delete (dkey + delmin)
n	$\boxed{O(1)}$	• Meld [take 2 pqs and combine them into 1]
$O(1)$	$O(1)$	• findmin [~~delemin + insert~~]

↑ n = #elements in queue at any time

How fast?: usual: heap. "binary"   Fredman + Tarjan: better Fibonacci heaps. [Amortized bounds]

So: for each vertex in nbrhood of C maintain the wt of edge out (cheapest)

~~When~~ 1 deletemin for each vertex v

now: update with ~~least~~ all edges from v to rest : d(v). decreasekeys

⇒ total time: $O(n \log n) + O(m)$.

Using binary heaps: $O(m \log n)$.

Boruvka : For each vertex, pick cheapest edge out.

// Unique edge wts ensure will pick a forest. (also undirected). F.

Contract the edges in F to get new graph G'. Repeat.

Fact: # of components in (G,F) = # vertices of $G' \le \frac{|V(G)|}{2}$.

$\Rightarrow$ #rounds $= O(\log n)$.

time per round : scan all the edges in G_i : time $O(m_i)$. to get F_i

Contract $G_i \longrightarrow G_{i+1}$: time $O(n_i + m_i)$ [HW1].

$\Rightarrow O(m \log n)$. $\longleftarrow$ Could be worse than Prim + Fib Heap? Yes.

$\longleftarrow$ Could be ~~worse than~~ $O(n^2 \log n)$? $\downarrow$ ~~even~~

[HW1]: Boruvka $\le O(n^2)$ time.

$\le O(m(\log(n^2/m)+1))$.

Fredman + Tarjan : $O(m \log^* n) \longrightarrow$ better than sorting $\longleftarrow$ Yao's $O(m \log\log m)$ in HW.

Actually $O(m\,\beta(m,n))$ where

$\beta(m,n) = \min\{i : \log^{(i)} n \le \frac{m}{n}\}$

$\simeq \log^* n - \log^*(m/n)$.

- for dense graphs $\beta(m,n) = O(1)$.
- for sparse graphs $\beta(m,n) \simeq \log^* n$.

Very simple idea. When graph is ~~dense~~ sparse the $O(m + n\log n)$ of Prim is doing a lot of more work in $n \log n$ than on the $O(m)$. so "balance it out"?

Start a Prim at vertex. if heap gets "too big", stop.

Recall : main bottleneck is delete min $O(\log \#\text{size of heap})$.

So if heapsize $\le K$ then pay $O(\log K)$ per extractmin total cost if we keep doing this :-

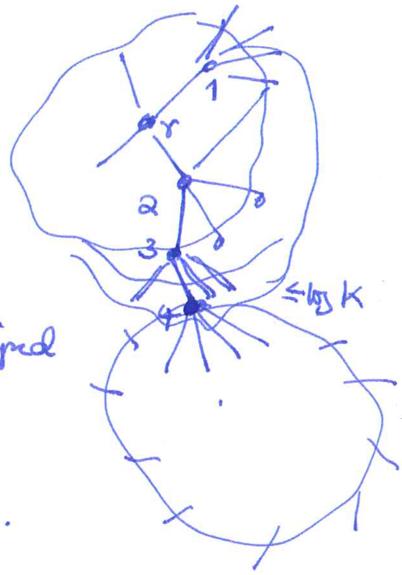
· each ~~vert~~ edge explored ≤ 2 times (from each end) ← $O(m_i)$ decreasekeys

· each ~~[scribble]~~ tree does deletemins as long as size of heap $\leq K$

⇒ total # extractmins $\leq O(n) \times \log K$.

N.b. { may meld heaps together if they meet — $O(1)$ time.

· in this case stop the process. even if the size of nbrhood has dipped below K. }



· Crucial fact: each ~~[scribble]~~ deletemin pays $O(\log k)$.

· How many trees eventually? Bdry of each component $\geq K$ ⇒ # trees $\leq \frac{2m}{K}$.

So from G_i with m_i edges, n_i nodes

↓

G_{i+1} with m_{i+1} edges. $n_{i+1} \leq \frac{2m}{K_i}$

What is K_i? set k_i st ~~$O(m) = O(n_i \log k_i)$~~ ⇒ $k_i \approx 2^{2m_i/n_i}$

$2m_i = n_i \log k_i$ ← $O(m)$ work per round.

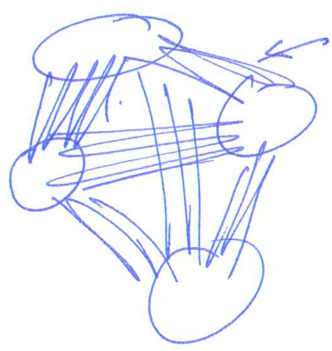
⇒ n_i nodes, now $n_{i+1} \leq \frac{2m}{2^{2m/n_i}}$ ⇒ $k_{i+1} = \frac{2m}{n_{i+1}} \geq 2^{k_i}$

⇒ K_i increases exponentially ⇒ ~~[scribble]~~ → $\beta(m,n)$.

$2m/n \to 2^{2m/n} \to 2^{2^{2m/n}} \ldots\ldots$ in $\log^* n$ rounds $k \geq n$.

Q: What about mi folly ; can we use that to get improvements?

Just by greedy we may not remove edges fast enough.

Eg.  ← very heavy edges here. All the work happens in the components, not across it.

We should use the red rule (cycle rule) to kill some of the edges. But that is next time.

Can save another log, and get $O(m \log \beta(m,n))$ runtime, but won't do it here.

History after this: KKT, (randomized) and then

Chazelle (97) $O(m\, \alpha(m,n))$ ← initially with an extra $\log \alpha$.

uses [soft heaps] ←→ may corrupt some of the keys in the heap!

(2000) Pettie & Ramachandran: an optimal MST algorithm:

Shows that if $MST^*(m,n)$ is optimal decision tree complexity of MST on m edge n node graphs, then

explicit algorithm that takes $O(MST^*(m,n))$ time on the pointer machine model.

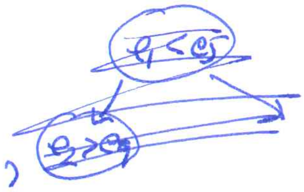
on all m ~~node~~ edge n vertex graphs.

Big Q: does ∃ a deterministic $O(m+n)$ time MST algorithm?

Enough to show that $MST^*(m,n) = O(m)$.

↑ ~~just a tree that says~~

just counts # of comparisons of edges done by algo?



Aside: (a) models of computation.

(b) is amortization necessary for (i) set Union find
(ii) Fibonacci heaps?

Union find: a lower bound in the cell probe model that need $m\alpha(m,n)$ time for m finds over n elements [Alstrup et al.]

Also if worst case complexity, can get $\Omega(\log n / \log\log n)$ in cell probe model $w = \log n$. [Fredman Saks].

Fibonacci Heaps: [Brodal Lagogiannis Tarjan STOC 12] bounds match Fib heaps but in the worst case [and pointer machine]. improving complicated RAM-machine soln of Brodal [SODA 96].

~~[illegible]~~

- Cell probe: memory cells, size w, we pay only for memory accesses.
- Transdichotomous RAM. words of size w (typically $\geq \log n$). Each operation manipulates $O(1)$ words at a time; finite set of operations. Based on what OPs allowed:
  - Word RAM : ops standard to C : $+, -, *, /$, mod, shift etc.
  - AC^0 ram : all ops implementable in AC_0 (constant depth circuits)

Pointer Machine: Memory is a giant DAG. ~~[illegible]~~ with constant branching factor.

Input to operations is a pointer. No pointer arithmetic allowed.