Lecture 4 :

Distributed Algorithms vs. Sublinear time Algorithms

- Vertex Cover

Simulating Greedy Algorithms in Sublinear time

- maximal matching

Distributed Algorithms vs. sublinear time algorithms on SPARSE graphs

max deg $\leq d$

Again, sparse graphs: max degree d
adj list representation

A problem to solve:

Vertex Cover

$V' \subseteq V$ is "Vertex Cover" (VC) if $\forall (u,v) \in E$

either $u \in V'$ or $v \in V'$

VC Question: What is min size of VC?

Note: in deg $\leq d$ graph, $|VC| \geq \frac{m}{d}$ since each node can cover $\leq d$ edges

(VC is NP-complete but there is a poly time 2-multiplicative approximation)

Can you approximate V.C. in sublinear time?

multiplicative? no! graph with no edges $|VC|=0$
graph with 1 edge $|VC|=1$
$\Rightarrow$ can't distinguish these cases in sublinear time but must answer 0 in first case & >0 in second.

additive? hard! need some mult error
computationally hard to approx to better than 1.36 factor (maybe even 2)

Combination?

<u>def</u>. $\hat{y}$ is (α, ε)-approximation of soln value y for minimization problem if

$$y \le \hat{y} \le \alpha y + \varepsilon$$

allows mult + additive error

(analogous defn for maximization problems)

Some Backround on distributed Algorithms

- Network
  - processors } max degree d known to all
  - links
- Communication round
  - nodes perform computation on (input bits, history of received msgs, random bits)
  - nodes send messages to neighbors
  - nodes receive messages from nbrs

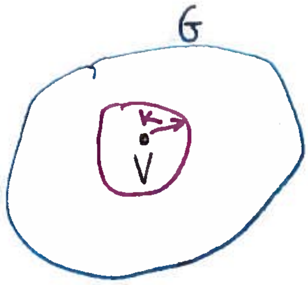
<u>def</u>. Vertex Cover problem for <u>distributed</u> <u>networks</u>:

(i.e. network computes on itself) not some other graph

- Network graph = Input graph
- at end, each node <u>knows</u> if in or out of VC (doesn't know about others necessarily)

Main insight on why fast distributed ⟺ sublinear time:

in k-round algorithm, output of node v only depends on nodes at distance at most k from v. At most d^k of these!



$\Rightarrow$ Can sequentially simulate v's view of distributed computation in $\leq d^k$ time & figure out if v in or out of VC * see next page

comment: if algorithm is randomized, v needs to know random bits (or be able to construct) of all d^k nbrs. ← must be consistent

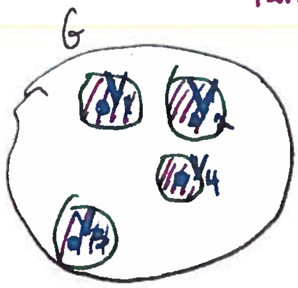
$\therefore$ fast distributed alg $\Rightarrow$ "oracle" which tells you if v is in VC

But are there fast VC distributed algorithms?

YES, will see some soon ↑ often called "local distributed algorithms"

How do you use this to approximate VC in sublinear time?

Parnas - Ron framework:



Sample nodes of graph $v_1 \dots v_r$

for each v_i, simulated distributed algorithm to see if $v_i \in VC$

Output $\dfrac{\# v_i\text{'s in VC}}{r} \cdot n$ } gives $\varepsilon \cdot n$ additive approx of VC which in turn is a c-multiplicative approx of c

Runtime $O(r \cdot d^{k+1}) \approx O(\frac{1}{\varepsilon^2} \cdot d^{k+1})$ (where k = # rounds of distributed alg, d = max degree of network)

Proof of correctness Chernoff/Hoeffding bnds

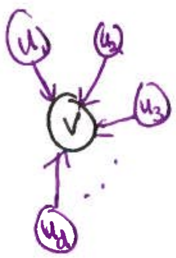
Simulating v's view of a k-round distributed computation:

round 0:

each node sends msg based on input + random bits

each node gets msg from each nbr which is based on their input, randombit

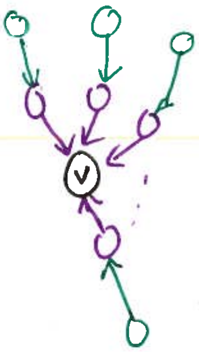
round 1:



each node sends msg based on { input, random bits, for input, randombits $\leq d$ nbrs + what they saw

each node gets msg based on nbrs info up to round 1,

round 2:



each node send msg based info of self + nbrs

each node receives msg based on nbrs + nbrs-of-nbrs

⋮

fast distributed algorithm for VC:

$i \leftarrow 1$

While edges remain:

- remove vertices of degree $\geq d/2^i$ + adjacent edges
- update degrees of remaining nodes
- increment i

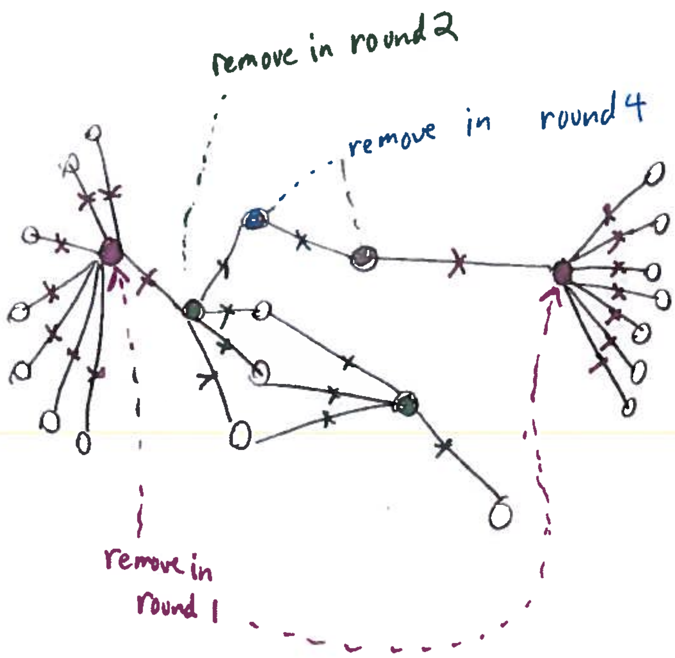
put these in vertex cover

Output all removed nodes as VC

#rounds: $\log d$

example:

$d = 8$



is it a VC?

no edges remain at end

all removed along with some adjacent vertex

Is it a good approximation?

Let Θ (optimal) be any min VC of graph

Thm $|\Theta| \le \text{output} \le (2\log d + 1)|\Theta|$

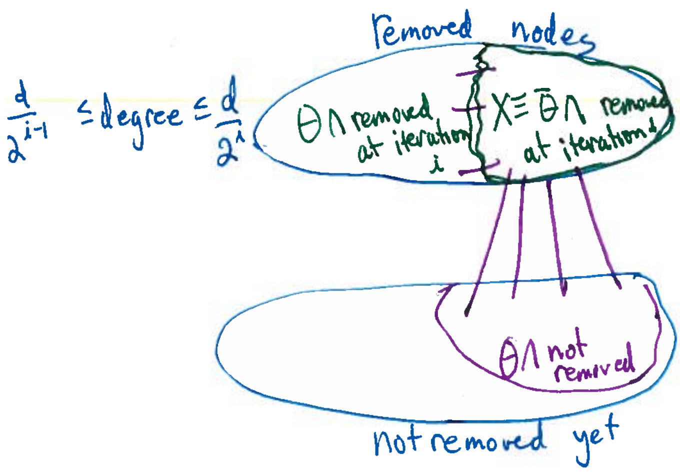
since output is VC ($|\Theta| \le$ output) — to prove ($\le (2\log d+1)|\Theta|$)

Proof

Claim: each iteration adds $\le 2|\Theta|$ new nodes to output V.C.

why?

observation: at i^{th} iteration
1) all nodes in graph have degree $\le \frac{d}{2^{i-1}}$ — anything bigger was removed earlier
2) all removed nodes have degree $\ge \frac{d}{2^i}$ — by algorithm design



Let X = removed at iteration i but not in Θ

note all edges touching X must also touch Θ at other end

why? Θ is a V.C.

edges touching X:

$$\geq \frac{d}{2^i} \cdot |X| \quad \text{since } \deg \geq \frac{d}{2^i}$$

$$\leq \frac{d}{2^{i-1}} |\Theta| \quad \text{since each edge has endpt in } \Theta,$$

+ each node in Θ has $\deg \leq \frac{d}{2^{i-1}}$

$$\Rightarrow \frac{d}{2^i} |X| \leq \frac{d}{2^{i-1}} |\Theta|$$

$$\Rightarrow |X| \leq 2|\Theta|$$

■ end-pf-of-claim

since $\leq \log d$ rounds,

$$\text{output} \leq |\Theta| + (2\log d)|\Theta| = (2\log d + 1)|\Theta|$$

■ end-pf-of-T

Gives $(O(\log d), \varepsilon)$-approx in $d^{O(\log d)}$ queries.

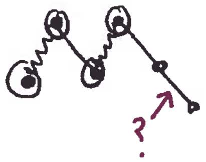
Can get $(2, \varepsilon)$-approx in $d^{O(\log(d/\varepsilon))}$ queries.

Sublinear Time Approximation Algorithms:

Estimating size of maximal matching in degree bounded graph

Why?

- relation to Vertex Cover



  - $VC \geq MM$ ← for each edge in matching, ≥ 1 endpt must be in VC
    these are disjoint
  - $VC \leq 2MM$ ← put all MM nodes in VC
    if an edge not covered, then violates maximality

- a step towards approx maximum matching

Note: if $\deg \leq d$, Maximal matching $\geq \frac{n}{d}$ ← to see this, run greedy algorithm

Greedy Sequential Matching Algorithm:

$M \leftarrow \phi$

$\forall\ e = (u,v) \in E$,
  if neither u or v matched,
    add e to M

Output M

output depends only on ordering of input edges

Observe:
M maximal, since if $\underset{(u,v)}{\overset{\shortparallel}{e}} \notin M$ either u or v alread matched earlier

Oracle reduction Framework

assume given deterministic "oracle" $\mathcal{O}(e)$
which tells you if $e \in M$ or not in one step

- $S \leftarrow s = \frac{7}{\varepsilon^2}$ nodes chosen iid.

- $\forall v \in S$

$$X_v = \begin{cases} 1 & \text{if any call to } \mathcal{O}((v,w)) \text{ for } w \in N(v) \text{ returns "yes"} \\ 0 & \text{o.w.} \end{cases}$$

- Output $\underbrace{\frac{n}{2s}}_{\text{since 2 nodes matched for each edge in } M} \sum_{v \in S} X_v + \underbrace{\frac{\varepsilon}{2} \cdot n}_{\text{makes an underestimate unlikely}}$

Behavior of output: Why does it work?

$$|M| = \frac{1}{2} \sum_{v \in V} X_v$$

$$E[|\text{output}|] = E\left[\frac{n}{2s} \sum_{v \in S} X_v\right] + \frac{\varepsilon}{2} \cdot n$$

$$= \frac{n}{2s} \sum_{v \in S} E[X_v] + \frac{\varepsilon}{2} n \quad \leftarrow \text{but } E[X_v] = \underbrace{\frac{2|M|}{|V|}}_{\text{fraction of matched nodes}} = \frac{2|M|}{n}$$

$$= \frac{n}{2s} \cdot s \cdot \frac{2|M|}{n} + \frac{\varepsilon}{2} n = |M| + \frac{\varepsilon}{2} n$$

$$\Pr\left[\left|\left(\frac{n}{2s} \sum_{v \in S} X_v + \frac{\varepsilon}{2} n\right) - E[\text{output}]\right| \geq \frac{\varepsilon}{2} n\right]$$

$$\|$$

$$\Pr\left[\left|\frac{n}{2s} \sum_{v \in S} X_v - |M|\right| \geq \frac{\varepsilon}{2} n\right] \leq \frac{1}{3} \quad \text{by additive Chernoff-Hoeffding}$$

■

Implementing the oracle:

Main idea: figure out "what would $\underbrace{\text{greedy}}_{\text{how?}}$ do on (v,w)?"

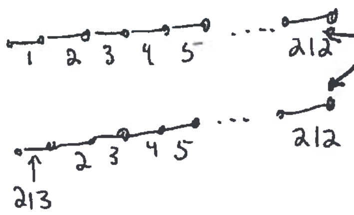
how? according to which input order?

do we need to figure out greedy decisions on all earlier node

Problem: Greedy is "sequential"

Can have long dependency chains

example:



even if you know the graph is a line, how do you know if edge is odd or even in the order?

How to implement oracle based on greedy?

To decide if e in matching,

- need to know decisions for adjacent edges that came before e in ordering
- do not need to know anything about any edge that comes after e in ordering since not considered by greedy algorithm before e

so, if any adjacent edge before e in ordering matched,

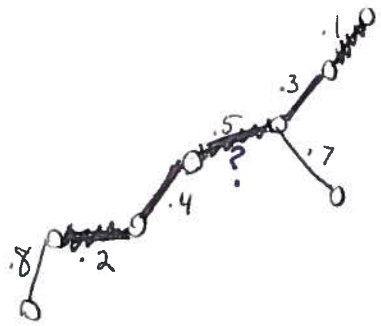
e is not matched

otherwise e is matched

How to break length of dependency chains?

assign random ordering to edges

example



is edge .5 in M?

- recurse on .3
  - recurse on .1
    - no other adjacent edges so
      .1 is matched
  - therefore .3 is not matched
  - no need to recurse on .7 since .5 < .7
- don't know yet about .5 so recurse on .4
  - recurse on .2
    - .8 comes after .2 in order
      so doesn't affect Greedy's behavior
    - same for .4
    - so .2 is matched
  - .4 is not matched
- .5 is matched

Implementation of oracle: assume ranks r_e assign to each edge e

to check if $e \in M$:

$\forall$ e' neighboring e,

- if $r_{e'} < r_e$, recursively check e' +

  if $e' \in M$, return "$e \notin M$" + halt

  else continue

return "$e \in M$"

↑ since no e' of lower rank than e is in M

Correctness: follows from correctness of greedy

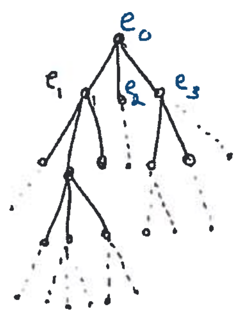
Query complexity:

Claim expected # queries to graph per oracle query is $2^{O(d)}$

Claim $\Rightarrow$ total query complexity is $\frac{2^{O(d)}}{\varepsilon^2}$

Pf of Claim

- Consider QueryTree where rootnode labelled by original query edge, Children of each node are edges adjacent to it.



- will only query paths that are monotone decreasing in rank

- $\Pr[\text{given path of length } k \text{ explored}] = \frac{1}{(k+1)!}$

- $\#$ edges in original graph at dist $\leq k$ in tree $\leq d^k$

- $E[\#\text{ edges explored at dist } \leq k] \leq \frac{d^k}{(k+1)!}$

- $E[\text{ total } \#\text{edges explored}] \leq \sum_{k=0}^{\infty} \frac{d^k}{(k+1)!}$

$$\leq \frac{e^d}{d}$$

- $E[\text{query complexity}] \leq d \cdot \frac{e^d}{d} = e^d = 2^{O(d)}$

∎