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 ≤ 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 ≤ d graph, $|VC| \geq \frac{m}{d}$ since each node can cover ≤ 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 $|VC| = 1$
} 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?

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

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

allows mult + additive error

(anulagous defn for maximization problems)

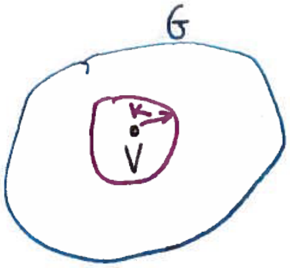
Some Backround on distributed Algorithms

- 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

def. Vertex Cover problem for distributed networks: (i.e. network computes on itself) ← not some other graph

- Network graph = input graph
- at end, each node knows 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!

⇒ Can sequentially simulate v's view of distributed computation in $\le d^k$ 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

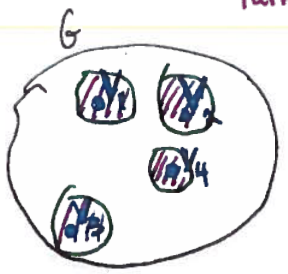
∴ fast distributed alg ⇒ "oracle" which tells you if v is in VC

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 $V_1 \ldots V_r$

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

Output $\frac{\# V_i\text{'s in VC}}{r} \cdot n$ } 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:

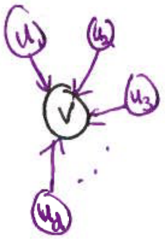
round 0:

each node sends msg based on input + random bits

ⓥ

each node gets msg from each nbr which is based on <u>their</u> input, random bit

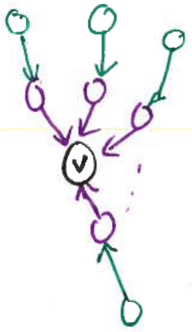
round 1:



each node sends msg based on { input, random bits, ≤d nbrs + what they saw for input, random bits

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:

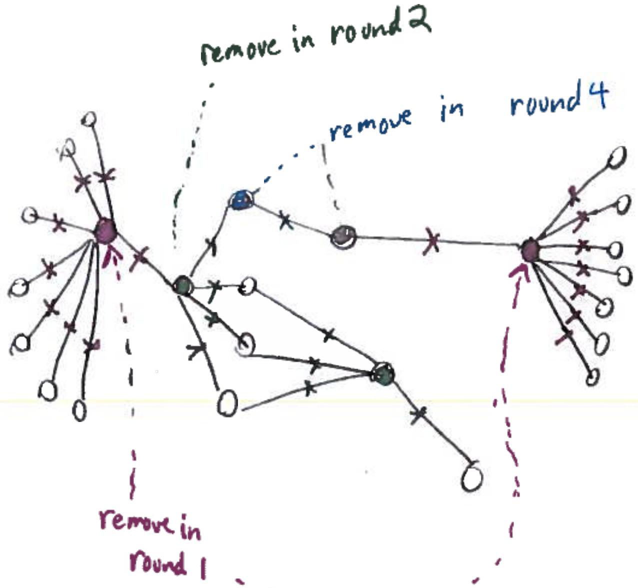
```
$i \leftarrow 1$
While edges remain:
    • remove vertices of degree $\geq d/2^i$ + adjacent edges
    • update degrees of remaining nodes
    • increment $i$
Output all removed nodes as VC
```

put these in vertex cover

#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 <u>any</u> min VC of graph

<u>Thm</u> $|\Theta| \leq$ output $\leq (2\log d + 1)\,|\Theta|$

↑ since output is VC ↑ to prove

<u>Proof</u>

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

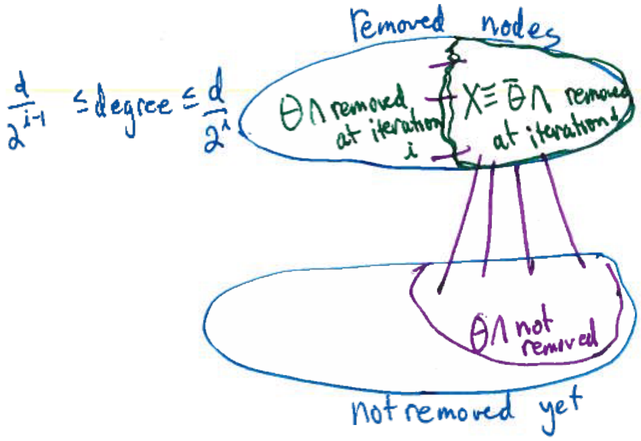
why?

Observation: at i^{th} iteration

1) all nodes in graph have degree $\leq \frac{d}{2^{i-1}}$ ⇙ anythin bigger was removed earlie

2) all removed nodes have degree $\geq \frac{d}{2^i}$

⇑ by algorithm design



Let X = removed at iteration i but <u>not</u> in Θ

<u>note</u> all edges touching X must also touch Θ at other end

why? Θ is a V.C.

$\#$ edges touching X:

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

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

output $\leq |\Theta| + (2\log d)|\Theta| = (2\log d + 1)|\Theta|$ ▨
end-pf-of-7

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

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

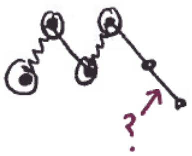
V0.1

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 ← φ
∀ e = (u,v) ∈ 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 $\underset{(u,v)}{e} \notin M$ either u or v alread matched earlier

No ②

Oracle reduction Framework

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

- $S \leftarrow s = \frac{8}{\varepsilon^2}$ 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 $\underbrace{\frac{n}{2s}}_{\substack{\text{since 2 nodes matched} \\ \text{for each edge in } M}} \sum_{v \in S} X_v + \underbrace{\frac{\varepsilon}{2} \cdot n}_{\substack{\text{makes an underestimate} \\ \text{unlikely}}}$

Behavior of output: Why does it work?

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

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

fraction of matched nodes

$$= \frac{n}{2s} \sum_{v \in S} E[X_v] + \frac{\varepsilon}{2} \cdot n \quad \leftarrow \text{but } E[X_v] = \frac{2|M|}{|V|} = \frac{2|M|}{n}$$

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

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

$$\shortparallel$$

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

■

No. 3

Implementing the oracle:

Main idea: figure out "what would greedy do on (v,w)?"

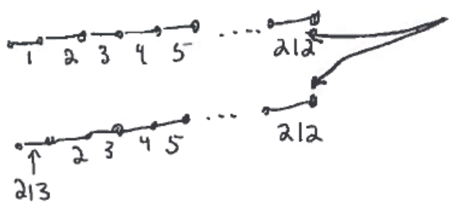
how? according to which input order?

do we need to figure out greedy's 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

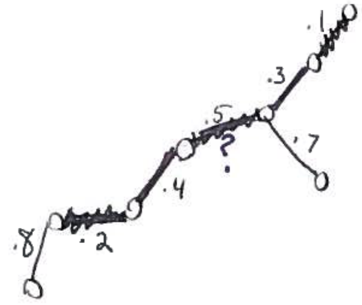
otherwise e is matched

NO.4

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

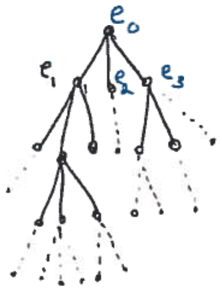
Query complexity:

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

Claim ⇒ 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 $\leq d^k$

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

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

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

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

▨