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' ∈ V is "Vertex Cover" (VC) if ∀ (u, v) ∈ E either u ∈ V' or v ∈ V'

VC Question: What is min size of VC?

Note: in deg ≤ d graph, |VC| ≥ m/d since each node can cover ∈ d edges

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

Can you approximate VC in sublinear time?

multiples?

graph with no edges |VC| = 0 => can't distinguish these cases in sublinear time

graph with 1 edge |VC| = 1

additive? hard! need some multiplicative error too; computationally hard to approximate better than 1.36 factor (maybe even 2)

Combination?
**def**: $\hat{y}$ is $(c, \varepsilon)$-approximation of solution value $y$ for minimization problem if

$$y \leq \hat{y} \leq cy + \varepsilon$$

allows multiplicative and additive error

(Analogous definition for maximization problems)

---

**Some Background on Distributed Algorithms**

- **Network**
  - processors \( \geq \) 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 neighbors

**def. Vertex Cover problem for distributed networks**: not some other graph

- Network graph = input graph (i.e., network computes on itself)
- At end, each node knows if in or out of VC (doesn't know about others necessarily)

**Main insight on why fast distributed \(\iff\) sublinear time**: in kround algorithm, output of node \(v\) only depends on nodes at distance at most \(k\) from \(v\). At most \(d^k\) of these!
Can simulate $V$'s view of distributed computation in $\leq d^k$ time. Figure out if $v$ is in or out of VC.

Comment: if algorithm is randomized, $v$ needs to know random bits (or be able to construct) of all $d^k$ neighbors. $k$ must be constant.

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 $V_1, \ldots, V_r$

For each $V_i$, simulate distributed algorithm to see if $V_i \in VC$.

Output $\#V_i's$ in VC/

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

Proof of correctness: Chernoff/Hoeffding bounds.
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, random bit.

**Round 1:**
- Each node sends msg based on $\exists$ input, random bits, ed nbrs & what they saw for input/random bit.
- Each node gets msg based on nbrs info up to round 1.

**Round 2:**
- Each node sends msg based info of self & nbrs.
- Each node receives msg based on nbrs & nbrs of nbrs.
Fast distributed algorithm for VC:

\[ i = 1 \]

While edges remain:
- remove vertices of degree \( \geq d/2^i \) and adjacent edges
- update degrees of remaining nodes
- increment \( i \)

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

\( \text{dist}(\vec{e}) \)
Is it a good approximation?

Let \( \Theta \) be any \( \min VC \) of graph

Thm \[ |\Theta| \leq \text{output} \leq (2\log d + 1) |\Theta| \]

since \( \text{output is VC} \) to prove

Proof

Claim: each iteration adds \( \leq 2|\Theta| \) new nodes to output VC

Why?

Observation: at \( i \)th iteration

1) all nodes in graph have degree \( \leq \frac{d}{2^i} \)

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

\( \frac{d}{2^{i+1}} \leq \text{degree} \leq \frac{d}{2^i} \)

Let \( X \) be removed at iteration \( i \) but not in \( \Theta \)

note all edges touching \( X \) must also touch \( \Theta \) at other end

why? \( \Theta \) is a VC.

\( \Theta \) not removed

Not removed yet

Removed nodes

\( \Theta \) removed at iteration \( j \)

\( \Theta \setminus \Theta \) removed at iteration \( j \)
# edges touching $X$:

\[ \approx \frac{d}{2^i} \cdot |X| \quad \text{since } \deg \geq \frac{1}{2^i} \]

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

\[ + \quad \text{each node in } \Theta \quad \text{has } \deg \leq \frac{d}{2^{i-1}} \]

\[ \Rightarrow \quad \frac{d}{2^i} |X| \leq \frac{d}{2^{i-1}} |\Theta| \]

\[ \Rightarrow \quad |X| \leq 2|\Theta| \]

\[ \text{end pf of claim} \]

since \leq \log d \quad \text{rounds,}

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

\[ \text{end pf of claim} \]

Gives \( O(\log d, \epsilon) \)-approx in \( d \log d \) queries.

Can get \( 2\epsilon \)-approx in \( d \log d / \epsilon \) queries.
Sublinear Time Approximation Algorithms:

Estimating size of maximal matching in degree bounded graph

Why?

- Relation to Vertex Cover
  - $VC \geq MM$ \(\Leftarrow\) for each edge in matching, at least one end node must be in VC (these are disjoint)
  - $VC \leq 2MM$ \(\Leftarrow\) put all MA nodes in VC
    - if an edge not covered, then violates maximality

- A step towards approximating maximum matching

Note: if $\deg u \geq d$, maximal matching $\geq \frac{n}{d}$ \(\Leftarrow\) to see this, run greedy algorithm

Greedy Sequential Matching Algorithm:

1. $M \leftarrow \emptyset$
2. $\forall e = (u,v) \in E$
   - if neither $u$ or $v$ matched, add $e$ to $M$

Output $M$

Observe:

- $M$ maximal, since if $e \notin M$ either $u$ or $v$ already matched earlier
Oracle Reduction Framework

- Assume given deterministic "oracle" \( O(e) \)
- which tells you if \( e \in M \) or not in one step

\[ s = \frac{p}{2^a} \]
- \( s \) nodes chosen iid

\[ \forall v \in S \]
- \( \chi_v = \begin{cases} 1 & \text{if any call to } O(y_w) \text{ for } w \in N(v) \text{ returns "yes" } \\ 0 & \text{o.w.} \end{cases} \]

- Output \( \frac{n}{2^a} \sum_{v \in S} \chi_v + \frac{\varepsilon}{2^a} \cdot n \)
- makes an underestimate unlikely

Since 2 nodes matched for each edge in \( M \)

**Behavior of output: Why does it work?**

\[ |M| = \frac{1}{2} \sum_{v \in V} \chi_v \]

\[ E[|\text{output}|] = E\left[ \frac{n}{2^a} \sum_{v \in S} \chi_v \right] + \frac{\varepsilon}{2^a} \cdot n \]

\[ = \frac{n}{2^a} \cdot s \cdot \frac{2|M|}{n} + \frac{\varepsilon}{2^a} \cdot n = |M| + \frac{\varepsilon}{2^a} \cdot n \]

\[ P_r \left[ \left| \frac{n}{2^a} \sum_{v \in S} \chi_v + \frac{\varepsilon}{2^a} \right| \geq \frac{\varepsilon}{2^a} \cdot n \right] \leq \frac{1}{3} \] by Chernoff-Hoeffding
Implementing the oracle:

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

Problem: Greedy is "sequential". Can have long dependency chains.

Example:

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 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 to .1
    * 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 $e \in M$:

1. $\forall e'$ neighboring $e$, $r_{e'} < r_e$, recursively check $e' +$

2. if $e' \in M$, return "$e \notin M"$ and 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
oracles query is $2^{O(d)}$

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

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

- Will only query paths that are monotone decreasing in rank.

\[ P \left[ \text{given path of length } k \text{ explored} \right] = \frac{1}{(k+1)!} \]

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

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

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

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