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 \textit{SPARSE} graphs

\[ \text{max degree } \leq d \]

Again, \textit{Sparse graphs}:

- \text{max degree } \leq d
- \text{adj list representation}

\[ \text{A problem to solve:} \]

\underline{Vertex Cover}

\[ V' \subseteq V \text{ is } "\text{Vertex Cover}\ (VC)" \text{ if } \forall \ (u,v) \in E \]

either \( u \in V' \) or \( v \in V' \)

\underline{VC Question:} What is \text{min size of VC}?

\[ \text{Note: in } \text{deg } \leq d \text{ graph, } \text{VC} \leq \frac{m}{d} \text{ since each node can cover } \leq d \text{ edges} \]

\( \text{VC is NP-complete, but there is a polytime } 2\text{-multiplicative approximation} \)

\text{Can you approximate VC in sublinear time?}

- multiplicative: \text{no!} graph with no edges \( |\text{VC}| = 0 \)
- graph with 1 edge \( |\text{VC}| = 1 \)

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

Combination?
**Definition**: \( \hat{y} \) is \((a, \epsilon)\)-approximation of solution value \( y \) for minimization problem if

\[
y \leq \hat{y} \leq ay + \epsilon
\]

(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

**Vertex Cover problem for distributed networks**: Let 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 \( \Rightarrow \) sublinear time**: In k-round algorithm, output of node \( v \)

- only depends on nodes at distance \( \leq k \) 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 $d^k$ nbors. $\tau$ must be consistent.

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 algorithm".

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 $\frac{\#V_i's \ in \ VC}{r}$ gives $\pm n$ additive approx of $VC$ which in turn is a $c$-multiplicative approx of $\epsilon$.

Runtime $O(r \cdot d^{k+3}) \approx O(\frac{c}{\epsilon^2} \cdot d^4)$ (where $K$ = # rounds of distributed alg $\backslash d$ = max degree of network).

Proof of correctness Chernoff/Hoeffding bounds.
Simulating $v_i$’s view of a $k$-round distributed computation:

**Round 0:**
- Each node sends $msg$ based on output + random bits
- Each node gets $msg$ from each nbr which is based on their input, randombit

**Round 2:**
- Each node sends $msg$ based on nbrs info up to round 1
- Each node gets $msg$ based on nbrs + what they saw for input, randombit
- Each node receives $msg$ based on nbrs + nbrs of_nbrs
A fast distributed algorithm for VC:

1. \( i = 1 \)
2. While edges remain:
   - remove vertices of degree \( \geq d/2 + \text{adjacent edges} \)
   - 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 vertices
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 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 \( \leq \frac{d}{2^{i-1}} \)
2) all removed nodes have degree \( \geq \frac{d}{2^i} \)

Let \( X \in \Theta \) removed at iteration \( i \)

not in \( \Theta \)

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

why? \( \Theta \) is a VC.
# edges touching $X$:
\[
\geq \frac{d}{2^d} \cdot 1 \geq 1
\]

since \( \text{deg} = \frac{d}{2^d} \)

\[
\leq \frac{d}{2^{d-1}} |\Theta|
\]

since each edge has endpt in $\Theta$,

ey each node in $\Theta$ has \( \text{deg} \leq \frac{d}{2^{d-1}} \)

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

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

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

since $\leq \log d$ rounds,

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

\[\text{end pf of 7}\]

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

Can get $(2, \varepsilon)$-approx in $d O(\log \varepsilon)$ 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, at least one endpoint must be in VC, edges 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 degree, maximal matching $\geq \frac{n}{d}$ to see this, run greedy algorithm

Greedy Sequential Matching Algorithm:

$M \leftarrow \emptyset$

$\forall e = (u,v) \in E,$

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

(output depends only on ordering of input edges)
Oracle Reduction Framework

Assume given deterministic "oracle" $\theta(e)$ which tells you if $e \in M$ or not in one step.

- $S \subseteq S = \frac{n}{\alpha s}$ nodes chosen iid.

- $\forall v \in S$
  $X_v = \begin{cases} 1 & \text{if any call to } \theta(v,w) \text{ for } w \in N(v) \text{ returns "yes"} \\ 0 & \text{otherwise} \end{cases}$

- Output $\frac{n}{\alpha s} \sum_{v \in S} X_v + \frac{\varepsilon}{2} \cdot n$

Since 2 nodes matched for each edge in $M$ makes an underestimate unlikely

Behavior of output: Why does it work?

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

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

$= \frac{n}{\alpha s} \sum_{v \in S} E[X_v] + \frac{\varepsilon}{2} \cdot n$ (but $E[X_v] = \frac{2|M|}{n} = \frac{2|M|}{n}$)

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

$\Pr\left[ |\frac{n}{\alpha s} \sum_{v \in S} X_v + \frac{\varepsilon}{2} \cdot n| \geq \frac{\varepsilon}{\alpha} \cdot n \right] \leq \frac{1}{3}$ by additive Chernoff-Hoeffding
Implementing the oracle:

Main idea: figure out "what would greedy do on <formula>"?

Problem: Greedy is "sequential"

Can have long dependency chains

Example:

```
1 2 3 4 5 ... a12
\----\----\----
  a13  a14  a12
```

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 \( e \) is 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

is edge 5 in M?

* recurse on .3
  * recurse on .1
    * no other adjacent edges go
      .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
      .50 .2 is matched
    * .4 is not matched
  * .5 is matched
Implementation of oracle: assume ranks re assign to each edge e to check if e ∈ M:

∀ e' neighboring e,

- if \( r_e > r_{e'} \), recursively check e'
- if \( e' \notin M \), return "e ∈ M" and halt
  else continue

return "e ∈ M"

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

Correctness: follows from correctness of greedy

Query complexity:

\[
\text{Claim } \text{expected # queries to graph per oracle query is } 2^{O(d)}
\]

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

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

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

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

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

- \( \leq \frac{e^d}{d} \)

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