

Lecture 5

*Lecturer: Ronitt Rubinfeld**Scribe: Oleksandr Sokolov*

1 Overview

Distributed algorithms vs. sublinear algorithms on *sparse* graphs, i.e., graphs with maximum degree at most Δ , where Δ is a constant.

We will see a way of designing lots of sublinear algorithms for sparse (average degree $O(1)$) graphs. The idea is to use known fast distributed algorithms, plus a reduction. In other words, the goal is to turn distributed algorithms into sublinear-time algorithms. We start with vertex cover.

2 Vertex Cover

Definition 1 $V' \subseteq V$ is a vertex cover (VC) if for all $(u, v) \in E$, either $u \in V'$ or $v \in V'$.

VC question: What is the minimum size of a VC?

Notes:

- In a graph of degree at most Δ ,

$$|VC| \geq \frac{m}{\Delta} \geq \frac{n \cdot \Delta_{ave}}{\Delta_{max}} \cdot \frac{1}{2},$$

since each node covers at most Δ edges. So in constant- Δ graphs, the VC is big.

- In general, VC is NP-complete.
- There is a poly-time 2-approximation via matching: take a maximal matching and put both endpoints of each matched edge into the VC. Claim: this is a VC.

3 Approximating Vertex Cover in Sublinear Time

Question: Can you approximate VC in sublinear time?

- **Multiplicative?** No! We can't distinguish a graph with 0 edges from a graph with 1 edge in sublinear time.
- **Additive?** Hard, since VC is NP-complete. We need some multiplicative error (at least 1.36; may be a factor of 2).

So we use a combination of multiplicative and additive error:

Definition 2 $\hat{y}$ is an (α, ϵ) -approximation of the solution value y for a minimization problem if

$$y \leq \hat{y} \leq \alpha \cdot y + \epsilon.$$

(There is an analogous definition for a maximization problem, with the definition reversed.)

Remark In our case of $\Delta = O(1)$, the additive error can be “hidden” in the multiplicative error.

4 Background on LOCAL Distributed Algorithms

LOCAL is a well-studied model of distributed computation.

- **Network:** processors and links, with maximum degree Δ (processor i talks to processor j over a link).
- **Communication round:**
 - nodes perform computation on their input bits, processor ID (assume unique), history of messages received in previous rounds, and random bits;
 - nodes send messages to neighbors;
 - nodes receive messages from neighbors.

Vertex cover on a distributed network:

- the network graph = the input graph;
- at the end, each node knows if it is in the VC. It doesn't necessarily know about the other nodes or the total size.

LOCAL vs. sublinear: in the sublinear setting we query the adjacency list of G . In the LOCAL model we limit the number of rounds, not messages.

5 LOCAL Distributed vs. Sublinear Time

Main insight: In a k -round LOCAL algorithm, the output of node v can only depend on nodes of distance at most k from v . There are at most Δ^k of these!

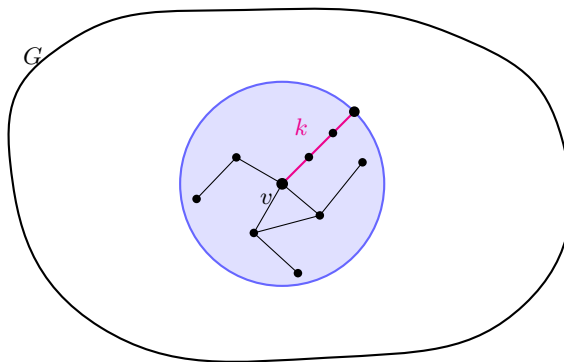


Figure 1: The nodes of distance at most k from v (shaded).

$\implies$ We can sequentially simulate v 's view of the distributed computation in at most Δ^k queries, and at the end know if v is in or out of the VC.

Remark If the LOCAL algorithm is randomized, v needs to know the random bits of all (at most Δ^k) neighbors, and these must be consistent.

So a fast distributed LOCAL algorithm for VC gives a fast sequential “oracle” for VC. Here “fast” means sublinear query complexity; what about sublinear time?

Is there a fast distributed LOCAL algorithm for VC? Yes, we will see one in Section 6.

What do we do with this oracle? Use it to estimate the *size* of the VC via sampling.

5.1 Estimating the Size of the VC: the Parnas–Ron Framework

Parnas–Ron framework

1. Sample nodes $v_1, \dots, v_r$ of the graph, with $r = 1/\epsilon^2$.
2. **for** each v_i : simulate the distributed LOCAL algorithm to see if $v_i \in VC$.
3. Output

$$\frac{\# v_i \text{'s in } VC}{r} \cdot n.$$

Runtime: $O(r \cdot \Delta^{k+1}) \approx O(\frac{1}{\epsilon^2} \cdot \Delta^{k+1})$, where k is the number of rounds of the distributed algorithm and Δ is the maximum degree.

Proof of correctness: Chernoff–Hoeffding bounds.

6 A Fast Distributed Algorithm for VC

Idea: a primal-dual based approximation algorithm, with matching as the primal and V.C. as the dual. We relate V.C. to the dual problem of *fractional matching*.

Definition 3 A fractional matching assigns weights $x_e \in [0, 1]$ to edges e such that for all v ,

$$\sum_{e \ni v} x_e \leq 1,$$

where the sum is over the edges containing v as an endpoint. The value of the fractional matching is $\sum_e x_e$.

In an integral matching, $x_e \in \{0, 1\}$ and $\sum_{e \ni v} x_e$ is either 0 (unmatched) or 1 (matched).

6.1 The Algorithm

- Initialize $x_e = 1/\Delta$ for every edge e .
- **for** $i = 1$ **to** $\log_{1+\gamma} \Delta$:
 - **for** each node v such that $\sum_{e \ni v} x_e \geq \frac{1}{1+\gamma}$:
 - * add v to the V.C.;
 - * freeze x_e for all $e \ni v$ (v sends a “frozen” message to all neighbors).
 - **for** each unfrozen edge e : set $x_e \leftarrow (1 + \gamma) x_e$.

At the end, any remaining v are *not* in the V.C.

Comments:

- Initially, the sum around each node is at most 1: we start with a trivial solution to matching (the primal).
- The unfrozen x_e gradually increase until the matching becomes maximal.
- **Number of rounds:** $1 + \log_{1+\gamma} \Delta$.
- **Time to simulate:** $\Delta^{\log_{1+\gamma} \Delta}$. If Δ is a constant, this is fine.

6.2 Why is it a V.C.?

If e is frozen, then at least one endpoint is in the V.C. Else, if e survives until the end,

$$x_e = (1 + \gamma)^{\log_{1+\gamma} \Delta} \cdot \frac{1}{\Delta} = \frac{\Delta}{\Delta} = 1$$

$\implies$ the endpoints would join the V.C. $\implies$ no e survives. So either one of u, v was frozen earlier, or both are frozen now. $\implies$ At least one endpoint is in the V.C. ■

6.3 Why is it a Small V.C.?

Note: our algorithm finds a fractional matching. Why? If v doesn't freeze its edges, then

$$\sum_{e \ni v} x_e < \frac{1}{1 + \gamma},$$

so in the next round

$$\sum_{e \ni v} x_e \leq \left(\frac{1}{1 + \gamma} \right) \cdot (1 + \gamma) = 1$$

$\implies$ it gets frozen before getting bigger than 1.

The weight of our fractional matching is $\sum_e x_e$.

Plan: show

1. (any) fractional matching value $\leq$ min (integral) V.C.;
2. (our) output V.C. $\leq 2(1 + \gamma) \cdot$ (our) fractional matching value.

$\implies$ Output V.C. $\leq 2(1 + \gamma) \cdot$ min V.C.

Lemma 4 Any fractional matching value $\leq$ min V.C. (In particular, this holds for the max fractional matching and for the one found in our algorithm.)

Recall: integral max matching $\leq$ min V.C., since each edge in the matching contributes at least 1 node to any V.C.

Proof Given a valid fractional matching with value $\sum_e x_e$ and an optimal V.C. V_{OPT} : for every e , assign x_e to an endpoint of e in V_{OPT} (if both endpoints are in V_{OPT} , choose arbitrarily). Then the total weight assigned to nodes is $\sum_e x_e$, and each node in V_{OPT} gets at most 1, since the weights come from a fractional matching. So

$$|V_{OPT}| \geq \frac{\text{total wt}}{\text{max wt per node}} \geq \frac{\sum_e x_e}{1} = \sum_e x_e.$$

■

Lemma 5 Output V.C. $\leq 2(1 + \gamma) \cdot$ (our) fractional matching value.

Proof Given the output V.C. $\hat{V}$: for every $u \in \hat{V}$, assign value 1 (so the total value is $|\hat{V}|$). Split u 's value among its neighboring edges proportional to x_e (from the algorithm): each edge e' gets

$$\frac{x_{e'}}{\sum_{e \ni u} x_e} \leq \frac{x_{e'}}{\frac{1}{1 + \gamma}} = (1 + \gamma) x_{e'}$$

from u . It can also get at most $(1 + \gamma) x_{e'}$ from its other endpoint, so the total assigned to e' is at most $2(1 + \gamma) x_{e'}$. Therefore

$$\text{total value} = |\hat{V}| \leq 2(1 + \gamma) \sum_e x_e,$$

where $\sum_e x_e$ is the weight of our fractional matching. ■