**0368.416701 Sublinear Time Algorithms** November 2, 2009

Lecture 3

*Lecturer: Ronitt Rubinfeld* *Scribe: Margarita Vald & Anat Ganor*

# Introduction

In this lecture we explore the connection between local distributed algorithms and sublinear time algorithms on sparse graphs. We will see how a round-efficient distributed algorithm may be used to construct a sublinear time algorithm.

Similar methods work for a range of problems that can be referred to as positive linear programming (LP) problems and deal with covering and packing questions (such as vertex cover, matching and dominating set). These problems, when working on sparse graphs, have local distributed algorithms (that use only few rounds).

This lecture will focus on an algorithm for approximating the vertex cover problem.

**Definition 1** (Vertex cover)**.** *Given $G = (V, E)$, $V' \subset V$ is a vertex cover if $\forall e = (u, v) \in E$ either $u \in V'$ or $v \in V'$.*

## Approximating Minimum Vertex Cover using Linear Programing

**Definition 2** (Integer Programming)**.** *Given an objective function, and a collection of linear constraints, find an assignment of 0-1 values to the variables such that all linear constraints are satisfied and the objective function is optimized.*

### Vertex Cover as Integer Constraints:

For a graph $G = (V, E)$ , we have variables $X_i$, which will be either 0 or 1, indicating whether vertex $i$ is part of the cover.

$$\begin{array}{lll} Minimize & \sum_{i=1}^{n} X_i & \\ subject\ to & X_i + X_j \geq 1 & for\ each\ (i,j) \in E \\ & X_i \in \{0,1\} & for\ each\ i \in V \end{array}$$

Integer programming is NP-complete and therefore we will consider the linear programming relaxation:

$$\begin{array}{lll} Minimize & \sum_{i=1}^{n} X_i & \\ subject\ to & X_i + X_j \geq 1 & for\ each\ (i,j) \in E \\ & X_i \in [0,1] & for\ each\ i \in V \end{array}$$

*The LP Rounding Algorithm*

- Solve LP relaxation optimally to get fractional solution $x^*$
- Round the fraction values to obtain a solution to the vertex cover problem, i.e., $S = \left\{i |\ x_i^* \geq \frac{1}{2}\right\}$
- Return $S$

**Claim 1.** *The set $S$ obtained by rounding the LP-relaxation is a vertex cover.*

**Claim 2.** *The vertex cover $S$ returned by the LP rounding is at most 2-times the optimal cover.*

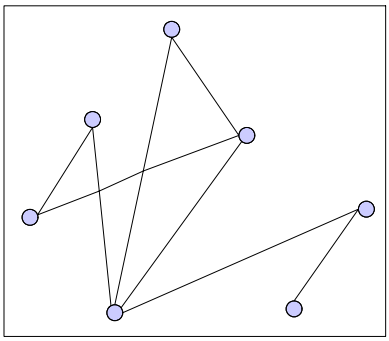

Figure 1: Communication network graph

We will work on sparse graphs and assume that $d$ is the maximum degree and that the average degree $\overline{d}$ is $\Omega(d)$. Since on such graphs the cover is large (approximately $\frac{n}{2}$ vertices), every vertex that is chosen at random is likely to be in the cover.

**Definition 3.** *$y'$ is $(\alpha, \epsilon)$-estimate of $y$ if $y \leq y' \leq \alpha y + \epsilon \cdot n$ (for $\alpha > 1$ and $\epsilon < 1$)*

We will see a sublinear time algorithm that computes an $(2 \log d + 1, \epsilon)$ approximation for vertex cover. Note that this approximation isn't very good when $\overline{d}$ is large.

### Our assumptions

We will assume the following:

- Our input graph will be a communication network, where every node is a processor and every edge is a line of communication (see Figure 1)
- The network is synchronous. In each round, each processor talks to each of its neighbors
- The graph is sparse with maximum degree $d$ and average degree $\overline{d} = \Omega(d)$
- Every processor knows $d$
- After $k$ rounds each processor decides if it's in the cover or not

## Simple Case

First we will assume that there's a deterministic distributed algorithm $\mathcal{D}$ that can solve the vertex cover problem in only $k$ rounds. We will see a sublinear "Oracle Reduction Algorithm" that estimates the cover size based on this assumption.
The new algorithm is a reduction from $\mathcal{D}$. It uses an oracle which computes the output of $\mathcal{D}$ and for every query on a vertex $v$ tells if $v$ is in the cover or not. Since the oracle can answer using $k$ rounds, the answer can be affected only by the vertices and edges that can be reached from $v$ during $k$ steps. We will call this sub graph the "**k neighborhood**" of $v$.

**Theorem 3.** *Given a deterministic distributed algorithm $\mathcal{D}$ that in $k$ rounds computes exactly a vertex cover $\mathcal{C}$, there's a sublinear algorithm with query complexity $O(\frac{d^{k+1}}{\epsilon^2})$ that computes $c$ s.t. $|\mathcal{C}| \leq c \leq |\mathcal{C}| + \epsilon n$.*

**Proof**

*The Oracle Reduction Algorithm (with oracle access to $\mathcal{D}$)*

- *Generating $S$*: pick $s = \frac{8}{\epsilon^2}$ nodes i.i.d from $[n]$
- $\forall v \in S$, let $G_k(v)$ be the "$k$ neighborhood" of $v$
  - query $\mathcal{D}(G_k(v))$
  - Define $X_v = \begin{cases} 1 & \text{if } \mathcal{D}(G_k(v)) \text{ adds } v \text{ to cover} \\ 0 & o.w. \end{cases}$
- Output $c = n \cdot \left(\frac{1}{s} \cdot \sum_{v \in S} X_v\right) + \frac{\epsilon n}{2}$

**Correctness**
The $X_v$'s cover is identical to $\mathcal{C} = \mathcal{D}(G)$, i.e., $X_v = 1$ iff $v \in \mathcal{C}$. Also, we have $E[X_v] = \frac{1}{n} \cdot \sum_{v \in V} X_v$ which can be approximated using $\frac{1}{s} \cdot \sum_{v \in S} X_v$
This way we can estimate $|\mathcal{C}| = n \cdot E[X_v]$
By the additive Chernoff bound we have:

$$Pr\left[\left|\frac{1}{s} \cdot \sum_{v \in S} X_v - E\left[X_v\right]\right| \geq \frac{\epsilon}{2}\right] \leq \frac{1}{3}$$

**Query complexity**
The query complexity of the algorithm is $O(\frac{d^{k+1}}{\epsilon^2})$ where $\frac{1}{\epsilon^2}$ is the number of times we look at $G_k(v)$ for some $v$ in $S$ and $d^{k+1}$ is an upper bound on the number of vertices in $G_k(v)$.
Note that every call to the oracle is considered one step of computation and we ignore the size of the messages. ■

**Remark**

- Adding $\frac{\epsilon n}{2}$ is just a shift to get $|\mathcal{C}| \leq c \leq |\mathcal{C}| + \epsilon n$ instead of $|\mathcal{C}| - \frac{\epsilon n}{2} \leq c \leq |\mathcal{C}| + \frac{\epsilon n}{2}$
- To reduce the error range we can run the same algorithm many times and return the median (see homework 0, question 2)

# General Case

The algorithm we used in the simple case to solve vertex cover in polytime doesn't exist. Therefore, we will use a randomized distributed algorithm $\mathcal{D}$ which gives an approximation. Although $\mathcal{D}$ is randomized and returns an approximation, the "Oracle Reduction Algorithm" can work the same way as before. To do that, we have to make sure all of $\mathcal{D}$ answers are consistent, so we must ensure that each processor $w$ uses the *same* random bits in every computation of $\mathcal{D}((G_k(v))$ for which $w$ is in the $k$ neighborhood of $v$.

## A Distributed Approximation Algorithm for Vertex Cover

The following is a distributed randomized algorithm. On every step it adds to the cover a vertex chosen at random out of the remaining vertices. We will show that this algorithm gives a multiplicative approximation and use it as the oracle of the "Oracle Reduction Algorithm".

*Distributed Approximation Algorithm for Vertex Cover (assume maximum degree d)*

- $\mathcal{C} \leftarrow \emptyset$
- For $i = 1$ to $\log d$ do:
  - Each vertex $v$ with degree $\geq \frac{d}{2^i}$ adds itself to $\mathcal{C}$
  - For all added $v$ remove $v$ and all adjacent edges from $V$
  - Update degrees of all remaining vertices
- Output $|\mathcal{C}|$

**Theorem 4.** *Let $VC_G$ be the optimal solution for the vertex cover problem. The algorithm above gives a multiplicative approximation of $(2\log d + 1)$, i.e. $VC_G \leq |\mathcal{C}| \leq (2\log d + 1)VC_G$*

**Proof** Let $G = (V, E)$ be a graph with maximum degree $d$.

**LHS:** Since at the end of run no edges remain and edge is removed only when one of its nodes is inserted into $\mathcal{C} \Longrightarrow \mathcal{C}$ is a vertex cover

**RHS:** Let $\theta$ be a minimum vertex cover. therefore, $|\theta| = VC_G$ and $\bar{\theta} = V \setminus \theta$

**Claim 5.** *In each iteration, the number of new vertices from $\bar{\theta}$ that added to $\mathcal{C}$ is $\leq 2 \cdot VC_G$*

**Proof** Let $Y_i$ be the vertices considered in iteration $i$. (i.e., current degree between $\frac{d}{2^{i-1}}$ and $\frac{d}{2^i}$). Let $X_i = Y_i \cap \bar{\theta}$.

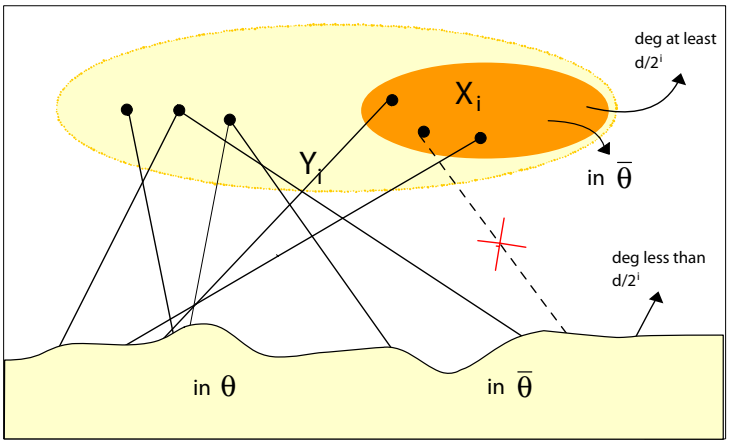


Figure 2: The partition of the vertices in the $i$th iteration

Note that all edges from $X_i$ must go to $\theta$ since $\theta$ is a vertex cover (see Figure 2). Therefore, $|X_i| \cdot \frac{d}{2^i} \leq |\theta| \frac{d}{2^{i-1}}$, which implies that $|X_i| \leq 2 \cdot |\theta| \leq 2 \cdot VC_G$ ■

By the claim we obtain that $|\mathcal{C}| \leq (2\log d + 1)\, VC_G$.

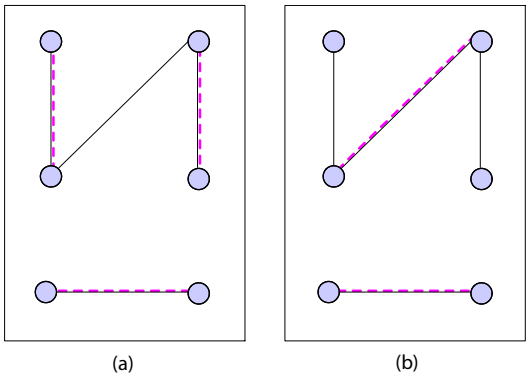


Figure 3: (a) Maximum matching (b) Maximal matching

■

**Corollary 6.** *Using the algorithm above as the oracle of the "Oracle Reduction Algorithm" yields a* $(2\log d + 1, \epsilon)$ *approximation for vertex cover using* $\frac{d^{O(\log d)}}{\epsilon^2}$ *queries*

## Maximal Matching

**Definition 4** (Matching)**.** *Given a graph* $G = (V, E)$*, a* matching $M$ *in* $G$ *is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex.*

**Definition 5** (Maximal matching)**.** *A* maximal matching *is a matching* $M$ *of a graph* $G$ *with the property that if any edge not in* $M$ *is added to* $M$*, it is no longer a matching.*

**Definition 6** (Maximum matching)**.** *A* maximum matching *is a matching that contains the largest possible number of edges. There may be many maximum matchings.*

Figure 3 depicts examples of maximum matching and maximal matching.

**Remark**

- Every maximum matching is maximal, but not every maximal matching is a maximum matching.
- The size of maximal matching is at most the size of the vertex cover.
- In graphs where $deg \leq d$ the size of maximal matching$\geq \frac{m}{2d-1}$.
  Proof: Each edge $e$ that is in the matching, exclude at most $2d-2$ other distinct edges (besides $e$ itself). Therefore, the number of edges in the matching is $\geq \frac{m}{2d-1}$.

*Greedy Sequential Maximal Matching Algorithm*

- $M \leftarrow \emptyset$

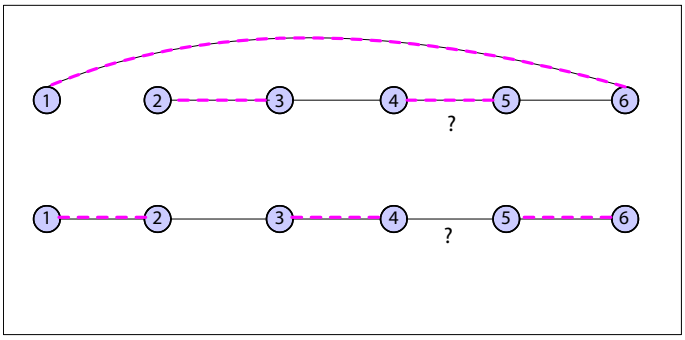


Figure 4: Implementing the oracle. The query $(4,5) \in M$ will cause a long dependency chain.

- $\forall e = (u,v) \in E$
  - if neither $u$ nor $v$ matched, add $e$ to $M$
- Output $M$

**Claim 7.** *$M$ is a maximal matching*

**Proof** Let $e = (u,v) \in E$ s.t $e \notin M$, then $u$ or $v$ matched. ■

## Implementing Oracle to Simulate Greedy

We would like to implement an oracle by simulating the above greedy algorithm.
To decide if an edge $e$ is in the matching :

- We must know if adjacent edges that come before $e$ in the ordering of edges are in the matching — if any are in the matching, $e$ is not; and only if all are not in the matching, the algorithm puts $e$ in the matching.
- Note that we do not need to know anything about edges that come after $e$ in the ordering.
- We want that the time that it takes to decide about $e$ will not depend on $n$.

The problem is that there might be long dependency chains (see Figure 4).

In the first case only the even edges are in the matching and in the second only the odd ones. When asked if edge $i$ is in the matching, the algorithm must check recursively all edges $1, 2, ..., i$ to get the answer. This requires $\Theta(i)$ recursion steps.

This can be solved by taking a random order of edges. When viewing the edges at a random order, the dependency chains break. The expected length of a dependency chain will not depend on $n$.

This is demonstrated in Figure 5: In this example we pick a random number between 0 and 1 for every edge to determine the order of the edges. When asked on edge 0.5 the algorithm will have to check only the edges 0.3 on the right side and $0.4, 0.2$ on the left side. The rest do not affect the answer.

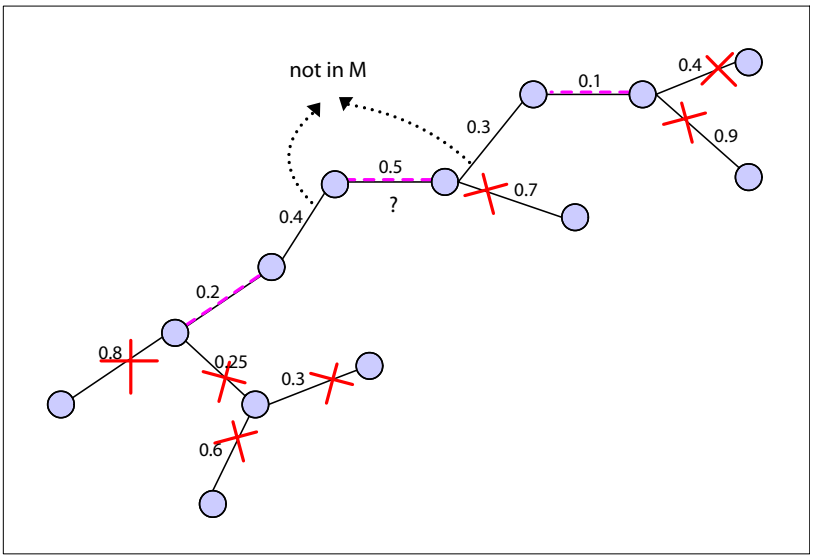


Figure 5: Improved oracle implementation