0368.416701 Sublinear Time Algorithms November 16, 2015

Lecture 5

Lecturer: Ronitt Rubinfeld Scribe: Nataly Brukhim, Roei Schuster, Brit Youngmann, Ariel Bruner

Lesson Overview

1. Estimating the weight of a Minimum Spanning Tree (MST) in a graph
2. Comparing distributed algorithms and sub-linear time algorithms
3. Presenting a distributed algorithm for approximating VC in constant time

1 Minimum Spanning Trees

Definition 1 *Given a connected, undirected graph, a* ***spanning tree*** *of that graph is a subgraph that is a tree and contains all the vertices of the graph.*

Definition 2 *Let T be a spanning tree of a weighted graph G. Let $w(T)$ be the sum of weights of all edges in T. We say that T is a* ***minimum spanning tree*** *(MST) if $w(T)$ is minimal.*

We will describe an algorithm that estimates the weight of a minimum spanning tree of a connected graph in sub-linear time.
Algorithm Input:

- A connected graph $G = (V, E)$ given in an adjacency list representation.
- $\forall (u, v) \in V^2, w_{uv} \in \{1, ..., \omega\} \cup \{\infty\}$, where $w_{uv} = \infty \iff (u, v) \notin E$.
- Maximum degree of vertices of G, d, and $w = max(w_{uv})$.

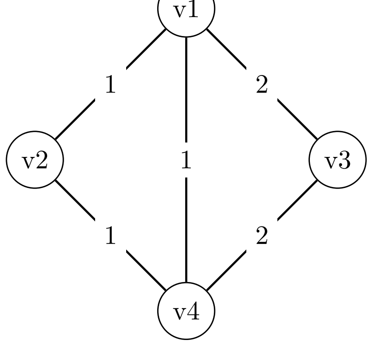
Algorithm Output: $\widehat{M}$ an estimation of M, the weight of an MST, such that $(1-\epsilon)M \leq \widehat{M} \leq (1+\epsilon)M$.
Note that the fact that G is connected implies that $(n-1) \leq w(T) \leq (n-1)\omega$.

We will see that the algorithm generates an additive rather than a multiplicative error. Once we have an additive estimate, we will see how to get a multiplicative estimate. The estimation of M will be done by first approximating the number of connected components in the following subgraphs:

Definition 3 *Let $G^{(i)} = (V, E^{(i)})$, where $E^{(i)} = \{(u, v) \in E | w_{uv} \in \{1, ..., i\}\}$, and $E^{(0)} = \phi$.*

Definition 4 *Let $C^{(i)}$ be the number of connected components of $G^{(i)}$.*

We will examine the following example:



In the example above $M = 4$. We can see that we need at least $|C^{(1)}|$ edges of weight 2 for every MST. So $M = 2(C^{(1)} - 1) + 1(n - 1 - (C^{(1)} - 1)) = n - 2 + C^{(1)}$. We will generalize this observation to the following claim.

Claim 5 $M = n - \omega + \sum_{i=1}^{\omega-1} C^{(i)}$

Proof Let α_i be the number of edges of weight i in any MST of G. Note that it can be proved that all MSTs have same α_i values. We use the following observation:

$$\sum_{i>l} \alpha_i = C^{(l)} - 1, \text{ where } C^{(0)} = n$$

We then get

$$\begin{aligned} M &= \sum_{i=1}^{\omega} i\alpha_i \\ &= \sum_{i=1}^{\omega} \alpha_i + \sum_{i=2}^{\omega} \alpha_i + ... + \sum_{i=\omega}^{\omega} \alpha_i \\ &= (n-1) + (C^{(1)} - 1) + (C^{(2)} - 1) + ... + (C^{(\omega-1)} - 1) \\ &= (n - \omega) + \sum_{i=1}^{\omega-1} C^{(i)} \end{aligned}$$

■

1.1 MST Approximation Algorithm

We will present an algorithm that will estimate $\sum_{i=1}^{\omega-1} C^{(i)}$, and hence obtain an estimation of M.

- For $i = 1$ to $\omega - 1$:
  - $\widehat{C}^{(i)}$ = approximation of $C^{(i)}$ to within $\frac{\epsilon}{2\omega}$ additive factor, with error probability $< \frac{1}{4\omega}$ (using $O(log\omega)$ runs of ApproximateCC).
- Output $\widehat{M} = n - \omega + \sum_{i=1}^{\omega-1} \widehat{C}^{(i)}$

Each call to *ApproximateCC* (shown in previous lesson), costs $O(\frac{d}{(\frac{\epsilon}{2\omega})^4})$. In total we get $O(\omega \frac{d}{(\frac{\epsilon}{2\omega})^4} log\omega) = \widetilde{O}(d\frac{\omega^5}{\epsilon^4})$. The probability that all calls to *ApproximateCC* give good approximations is atleast $1 - \frac{\omega}{4\omega} = 3/4$.
Hence, $|M - \widehat{M}| \leq \omega \frac{\epsilon}{2\omega} n = \frac{\epsilon n}{2}$ is the small additive error. Since $M \geq n - 1$, we get $|M - \widehat{M}| \leq \frac{\epsilon n}{2} \leq \frac{\epsilon}{2} 2M = \epsilon M$, the small multiplicative error.

2 A sense of the relationship between distributed algorithms and sub-linear time algorithms

In this section, we aim to show a correspondence between distributed and sub-linear algorithms. We will mainly consider tasks dealing with sparse graphs. The representation of the graphs will be held by adjacency lists. Assume that the given graph $G =< V, E >$ has maximum degree d.

2.1 Motivation

Our motivation is to explain the correspondence between fast distributed algorithms and sub-linear time algorithms.We will demonstrate this correspondence using the Vertex Cover (VC) problem. We introduce a sub-linear time solution of this task and a distributed type solution for it.

2.2 The Vertex Cover Problem

Definition 6 *Given an undirected graph $G =< V, E >$, a set of vertices $V' \subseteq V$ is called a Vertex Cover if $\forall(u, v) \in E, u \in V' \vee v \in V'$.*

2.2.1 Question:

Given graph G, What is the minimum size of a VC?

Observation 7 *For an undirected graph $G =< V, E >$ s.t. $deg(G) \leq d$, the minimal VC must contain at least m/d vertices, where $m = |E|$, since each vertex can cover at most d edges.*

This problems is NP-complete, yet there is a poly-time 2-mult approximation for this problem. We aim to approximate the size of the vertex cover in sub-linear time.

2.2.2 How can we approximate the minimum size of a VC?

- **Multiplicative:** no. For instance, consider two graphs of the same size, one with only one edge and another with no edges. In the first $|V'| > 0$ and for the second $|V'| = 0$. Such an algorithm must distinguish between both cases in sub-linear time (intractable).
- **Additive:** Hard. Need some multiplicative error. Computationally hard to do.

In order to solve this situation, we will take a combination of both (multiplicative and additive error factor).

Definition 8 *We refer to y' as an (α, ϵ)-estimate of solution y for a minimization problem if $y \leq y' \leq \alpha y + \epsilon$, i.e, we allow both multiplicative and additive errors at the same time.*

2.3 Background on distributed algorithms

In our context we use the following abstraction to distributed algorithms. We view distributed algorithms as networks with the following:

- Nodes: each node denotes a processor in the network.
- Links: each link connects between two processor units.
- Communication rounds: in each communication round, each of the processors sends messages to its neighbors.

The VC problem for distributed networks:

- Network graph: Input graph. The network computes on itself.
- Termination time: Each node knows if it is part of the VC or not.

2.4 Corresponding distributed and sub-linear time algorithms

We examine the VC problem, in a k-round algorithm, where the output of each node v depends only on nodes at distance at most k from v, i.e, at most d^k nodes. In other words, it simulates the view of v has of a k steps long distributed computation by using d^k steps, and figures out if v is in the VC or not.

In advance, we will introduce some VC distributed algorithms (often called local distributed algorithms). We will use this to approximate the size of the VC in sub-linear time. For this purpose we will first introduce the following framework -

2.5 Parnas-Ron framework:

- Sample r nodes from the graph: $v_1, .., v_r$.
- For each v_i (in the sample set), simulate a k steps long distributed algorithm to determine if $v_i \in$ VC or not.
- Output: $\frac{\# v_i \in VC}{r} n$

Query complexity: $O(rd^{k+1}) \approx O(\frac{1}{\epsilon^2} d^{k+1})$.

We will not introduce the entire proof of correctness, just notice it is based on Chernoff/Hoeffding bounds.

2.6 "Fast" distributed algorithm for solving the VC problem:

```
1. $i \leftarrow 1$
2. While edges remain:
    (a) Remove all vertices $v$ s.t $deg(v) > \frac{d}{2^i}$, and all adjacent edges.
    (b) Update degrees of remaining nodes.
    (c) Increment $i$.
3. Output: all the removed vertices.
```

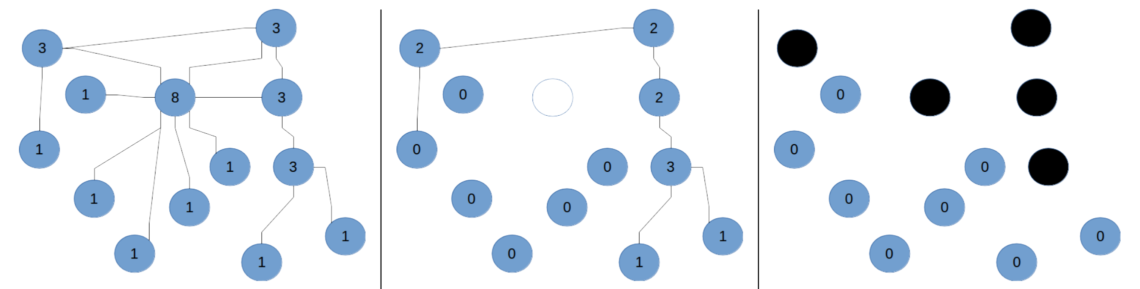
Lemma 9 *The output of the algorithm is a VC.*

Proof We remove an edge only when atleast one of its nodes is removed. The algorithm terminates when all edges are gone. ■

Lemma 10 *The algorithm requires up to* $\log d$ *rounds*

Proof it can only run until there are no edges left, and after $\log d$ rounds we remove all vertices of degree $\frac{d}{2^{\log d}} = 1$.Namely, any vertex with edges will be removed with its adjacent edges, and so the algorithm will terminate. ■

We will see an example for better understanding of how the algorithm runs:
The graph shown has a max degree of 8, so we start with removing the vertices of degree 4 and above. Next step would be removing the ones with degree 2 and above, and at this point - we are done. The vertices highlighted in black are the output. Looking carefully at the output, you can see it is not an optimal solution, as the minimum VC at this case is of size 3.



$i = 1$, remove $d \geq 4$	$i = 2$, remove $d \geq 2$	Done!

Note that this is a distributed algorithm, namely, each node is performing the algorithm on itself, updating its neighbors and getting updates from the neighbors. The processing can be done in parallel.

Theorem 11 *Let* $VC_{opt}(G) = min(\{|V'||V' is\ a\ VC\})$. *Then:* $VC_{opt}(G) \leq output \leq (2log(d)+1)VC_{opt}(G)$

Proof The first inequality holds by lemma 9. In order to prove the second inequality, we claim that in each iteration, we add at most $2VC_{opt}(G)$ vertices, plus some vertices from some minimal vertex cover θ. We will prove that in each iteration, the algorithm removes no more than $2VC_{opt}$ nodes that are not in θ. The rest follows by the bound on the number of iterations.
By construction, the degree of all vertices removed in iteration i is between $\frac{d}{2^i}$ and $\frac{d}{2^{i-1}}$ (as vertices with degree $\geq \frac{d}{2^{i-1}}$ were removed in previous rounds, and only vertices with degree $\frac{d}{2^i}$ or higher are removed in the current round). Note that the upper bound is on the degree of all vertices in the graph.
Now, we examine the vertices removed in iteration i of the algorithm.
Define $X \equiv$ All vertices that are not part of θ and removed in iteration i, and $\bar{X} \equiv$ All vertices that are part of θ and removed in iteration i. For each edge (v, u) touching X $(v \in X \vee u \in X)$, it must be either connected with an edge to another vertex $u \in \bar{X}$, therefore θ, or connected to a vertex that is removed in another iteration and belongs to θ (otherwise θ would not be a valid VC). Notice that it's not possible for any vertex in X to be connected to another vertex in X, as the meaning is that we have a vertex that is not covered by θ which must be a valid vertex cover.
Thus, the number of edges coming out of X is the number of edges going into θ removed at iteratoion i.
The lower bound on the number of edges coming out of X at iteration i is $|X|\frac{d}{2^i}$ (number of edges times the minimum degree of each vertex).
The upper bound on the number of edges removed in iteration i going to θ is $|\theta|\frac{d}{2^{i-1}}$ (number of vertices times the maximum degree).
Thus: $|X|\frac{d}{2^i} \leq |\theta|\frac{d}{2^{i-1}} \Rightarrow |X| \leq 2|\theta|$. ■