**6.896 Sublinear Time Algorithms** February 23, 2017

Lecture 4

*Lecturer: Ronitt Rubinfeld* *Scribe: Laura Gustafson*

## Overview

- K-round Distributed Algorithms
- Parnas-Ron Framework
- Distributed Sub-Linear Time Vertex Cover
- Oracle Reduction Framework
- Sub-Linear Time Maximal Matching

# 1 Relationship between Distributed and Sub-linear Algorithms

We want to take advantage of the work done in distributed algorithms and apply it to sub-linear algorithms. Why? We want to take advantage of the work we can do in a constant $k$ rounds.

## 1.1 K-round Distributed Algorithms

In distributed algorithms, there is no separate input network and distributing network. The network that you want to determine vertex cover on (for example) is the distributed network.

## 1.2 Vertex Cover

The goal for our algorithm is that after some constant $k$ rounds, every node will know whether or not it is a part of the vertex cover of the network. This algorithm will have $d^k$ query complexity where $d$ is the maximum degree. The first thing to consider is what would I send to my neighbors in 1 step (ie round 1). To determine this, I can look at my neighbors input and figure out what they would send. After two rounds, I could have received messages from my all of my neighbors and figured out how to combine what they sent me. After a total of $k$ steps, I can only receive mail from within a $k$-ball (i.e. all nodes within a $k$-radius of me). Any nodes outside of the $k$-ball do not influence me. Therefore, we only need to look within the $k$-ball of a node for vertex cover. For vertex cover, we will make $d^k$ queries, and will have an oracle that will let us compute whether or not I am in the vertex cover.

## 1.3 Parnas-Ron Framework

This distributed algorithms framework has been used for multiple problems (maximum matching, coloring problems). The idea is shown in Algorithm 1.

**Algorithm 1** Parnas-Ron Framework

**procedure** ParnasRonFramework
  *sample nodes* $v_1 ... v_r$
  **for all** $v_i$ **do**
    *simulate distributed algorithm to see if* $v_i \in VC$
  **Output:** $\frac{\# v_i \in VC}{r} \cdot n$

The run-time of this algorithm is $r \cdot d^{k+1}$. Using Chernoff bounds, we find that we will sample $r \approx O(\frac{1}{\epsilon^2})$.

### 1.4 Distributed Algorithm for Vertex Cover

*Note*: This is not the best sub-linear time algorithm for vertex cover. Remember that vertex cover is a NP-hard problem. This algorithm will not give us an optimal solution but instead a $2 * log(d_{max})$ approximation. Remember that we can also do a 2-approximation for vertex cover in polynomial time. Suppose our graph looks like Figure 1.

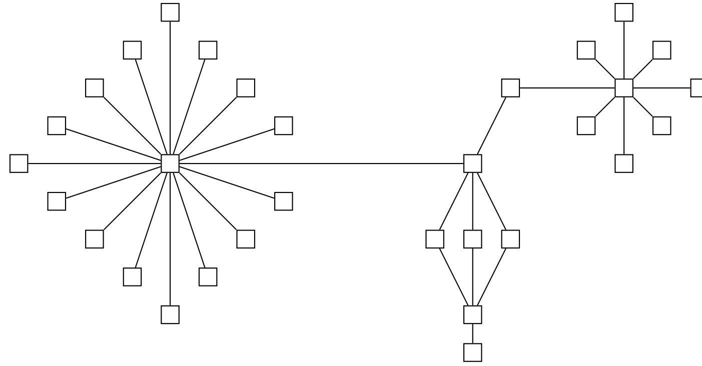

**Figure 1**: Example graph for vertex cover.

Our algorithm is a $log(d_{max})$ round distributed algorithm for Vertex Cover. It gives us a $2*log(d_{max})$ approximation, where $d_{max}$ is the maximum degree of any node in the graph.

**Algorithm 2** Distributed Vertex Cover

**procedure** DISTRIBUTEDVERTEXCOVER
  $i \leftarrow 1$
  $V^* \leftarrow \{\}$
  **while** *edges remain* **do**
    $V^* \leftarrow$ *nodes with degree* $> \frac{d}{2^i}$
    *Remove nodes with degree* $> \frac{d}{2^i}$
    $i \leftarrow i + 1$
  **Output:** $V^*$

In our example the max-degree is 16 (we can always assume it is an even power of 2).

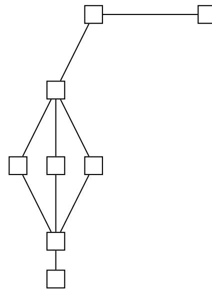

**Figure 2**: Graph after i=1 iteration.

After i=1, our graph will look like Figure 2. Because *maximum degree* = 16, and $16/2^1 = 8$, we removed all nodes with degree $\geq 8$.

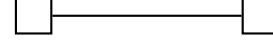

**Figure 3**: Graph after i=2 iteration.

After i = 2, our graph will look like Figure 4. We removed all nodes of degree $8 \geq d \leq 16/2^2$. Since everything with degree $\geq 8$ was already removed, we will remove remaining nodes with degree $\geq 4$.

For i=3, $16/2^3 = 2$. There are no nodes in the remaining graph (Figure 3) that have degree $\geq 2$, so we don't update the graph.

For i=4 $16/2^4 = 1$. The two remaining nodes have degree of 1, so we add both of them to the vertex cover.

Is the answer always a Vertex Cover? At the final time step, we are removing all nodes with degree $\geq 1$, so all edges are removed with at least one of their neighbors in the Vertex Cover. In the max degree = 1 case (ie the last time step), both nodes on one edge are added to the Vertex Cover as both have degree 1. So it IS a Vertex Cover.

Why is it a good Vertex Cover? We can prove that we remove a bound of *wasteful* (not in the ideal Vertex Cover) nodes.

**Theorem 1** $VC(G) \leq |V^*| \leq 2 \cdot \log(d) + 1 \leq VC(G)$

**Proof** $\theta$ = any minimum vertex cover such that $|\theta| = VC(G)$

**Claim:** in each iteration we add **at most** $2 \cdot VC(G)$ nodes not in $\theta$.

**Implies:** at the end we removed $\leq |\theta| + \log(d) \cdot 2 \cdot VC(G)$ which is equivalent to saying $\leq VC(G) + \log(d) \cdot 2VC(G)$ as $|\theta| = VC(G)$. We possibly removed $VC(G)$ nodes in $\theta$, and also an additional *extra* $2VC(G)$ nodes each round for $\log(d)$ rounds.

We at each time step, we can separate our nodes into two sets, nodes that we removed and nodes that we did not remove. We know all nodes that we removed at time step i have $\frac{d}{2^{i-1}} > degree \geq \frac{d}{2^i}$. We know all remaining nodes have $degree < \frac{2}{d^i}$. We also have $\theta$ and we want to figure out which nodes we removed that were not in $\theta$. We want to prove that the number of nodes not in $\theta$ is not that large.

If we have a node in our VC that is not in $\theta$, we know that it must point to something in $\theta$. We know that the item in $\theta$ has $degree < \frac{d}{2^i}$ (or else we would have removed it).

Let $X$ = nodes removed in round i that are not in $\theta$. Number of edges from $X$ to $\theta$ is $\geq |x| \cdot \frac{d}{2^i}$. The number of edges touching $\theta < \frac{d}{2^i - 1}|\theta|$ as $\frac{d}{2^i - 1}$ is the maximum degree in the graph. Likewise the number of edges touching $X \geq |X| \cdot \frac{d}{2^i}$, as each node in $X$ has degree at least $\frac{d}{2^i}$. The number of edges touching $X$ must be a subset of the edges touching $\theta$, as all edges that touch $X$ must touch $\theta$. Putting these two inequalities together, we get $|X| \cdot \frac{d}{2^i} \leq \frac{d}{2^{i-1}} \cdot |\theta|$. From this we get $|X| \leq 2 \cdot |\theta|$. We are now done with the proof. ■

# 2 Maximal Matching

We know that for a graph $VC \geq$ maximal matching. When we have a maximal matching all the nodes are disjoint. We need at least one of the nodes in the VC, meaning we need at least maximal-matching nodes in our vertex cover. We also know that $VC \leq 2 \cdot$ *maximal matching*, as if if we put all of the nodes touched by an edge in the maximal matching we by definition would have to have a vertex cover. If we do not, then we did not have a maximal matching.

## 2.1 Maximal Matching on Graphs with Degree at Most d

We know that for a graph $G$ with maximum degree $d$, the maximal matching $\geq \theta(\frac{n}{d})$. We know this as we can simply take out an edge and remove the 2 nodes adjacent to the edge and all edges adjacent to them until we are done.

**Algorithm 3** Greedy Maximal Matching

**procedure** GreedyMaximalMatching
  $M \leftarrow \{\}$
  **for all** $e \in (u, v) \in E$ **do**
    **if** *neither u or v was previously matched* **then**
      $M \leftarrow e$
  **Output:** $M$

This greedy algorithm is sequential and not sub-linear.
The only thing this greedy algorithm depends on is the order of edges.

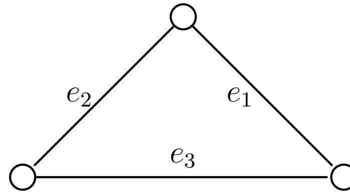


**Figure 4**: Example graph.

As you can see in Figure 4, the edges that will be in the matching are dependant on the order the greedy algorithm iterates through them. For example if $e_1$ is chosen first, then it will be the maximal matching. If $e_2$ is chosen first, it will be the maximal matching. And if $e_3$ is chosen first, it will be the maximal matching. There are never any edges where both endpoints aren't in the maximal matching.

## 2.2 Oracle Reduction Framework for Maximal Matching

We now want to estimate the size of the matching. Suppose we have a deterministic ORACLE which in one step can tell us whether or not an edge is in the maximal matching.

**Algorithm 4** Maximal Matching

**procedure** OracleReductionMaximalMatching
  $S \leftarrow \frac{8}{\epsilon^2}$ *independently at random chosen nodes*
  **for all** $v \in \mathcal{S}$ **do**
    **if** $ORACLE(v, w) ==$ *yes for any* $w \in N(V)$ **then**
      $x_v \leftarrow 1$
    **else**
      $x_v \leftarrow 0$
  **Output:** $n \cdot \frac{1}{2} \cdot \frac{\sum_{v \in S}(x_v)}{2 \cdot S} + \frac{\epsilon}{2} \cdot n$

The reason that we add a factor of $\frac{\epsilon}{2} \cdot n$ to our answer is to make under counting unlikely. The algorithm is outputting the fraction of samples matched. If we take enough samples, we know that it is a good estimate of the whole. The reason for the factor of $\frac{1}{2}$ is that the size of the matching is one half the number of nodes (in order to get edges between two nodes in the matching). In this algorithm the ORACLE has a maximal matching $M$ in its head.

**Claim:** $E[ouptut] = |M| + \frac{\epsilon}{2}$ where $|M|$ is the size of the matching in the ORACLE's head.
**Proof** $E[|output|] = E[\frac{n}{2 \cdot S} \sum_{v \in S} x_v] + \frac{\epsilon \cdot n}{2}$. This is equivalent to $\frac{n}{2 \cdot S} \sum_{v \in S} x_v + \frac{\epsilon \cdot n}{2}$. We know that the $E[x_v] = \frac{2 \cdot |M|}{|V|}$ and that $|V| = n$. The $E[x_v]$ is the fraction of matched nodes. This gives us

$E[|output|] = \frac{n}{2 \cdot S} \cdot S \cdot \frac{2 \cdot |M|}{n} + \frac{\epsilon \cdot n}{2}$, as expectation is linear. This reduces to $|M| + \frac{\epsilon \cdot n}{2}$. The probability $P(|\frac{n}{2 \cdot S} \sum_{v \in S} x_v] + \frac{\epsilon \cdot n}{2} - E[output]| \geq \frac{\epsilon \cdot n}{2})$ This is equivalent to $P(|\frac{n}{2 \cdot S} \sum_{v \in S} x_v] - |M|| \geq \frac{\epsilon \cdot n}{2})$. By additive Chernoff-Hoeffding bounds, we get $P \leq \frac{1}{3}$. ■

## 2.3 Maximal Matching Oracle

The question is now how do we make the ORACLE? It is non-trivially difficult to simulate the greedy algorithm.

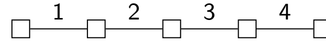


**Figure 5**: Graph where ordering is linear.

Take for example Figure 5. What would the greedy algorithm do about edge 4? That would depend on what the algorithm did on edge 3. That result depends on what the greedy algorithm did on edge 2 and onwards. In order to determine what the greedy algorithm would do for one edge, we need to follow the dependence chain all the way back to the beginning. In cases like Figure 6, this dependence chain could be linear in length.

In order to determine whether and edge $e$ is in the matching, we only care about edges that come before it in the ordering. We need to need to know the decisions about the edges that come before in the ordering. We do **not** need to know anything about the edges that come after $e$ in the ordering.

In order to simulate a greedy algorithm, we will use a random ordering of the edges. Each edge will receive a random ranking.

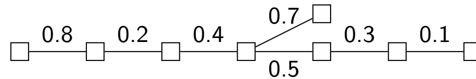


**Figure 6**: Example graph for maximal matching.

Suppose we are now looking at the edge with weight 0.5 in Figure 6. We take the following steps to determine if 0.5 is in the matching:

- In order to determine if it is in the matching, we first look at its neighbors. Two of its neighbors, 0.3 and 0.4, appear first in the ordering. We must first determine if either of those edges are in the matching. If they are, then 0.5 is not in the matching. If neither of 0.3 or 0.4 is in the matching, though, then 0.5 is in the matching.
- Now we will look at edge 0.3. Is it in the matching? To determine this, we must look at edge 0.1.
  - Looking at edge 0.1, it has no neighbors with rank less than it, so edge 0.1 **is** in the matching.
  - Going back to edge 0.3, we now know that 0.1 is in the matching, so edge 0.3 **is not** in the matching.
- Now we look at 0.5's other neighbor, 0.4. In order to determine if 0.4 is in the matching, we must know if 0.2 is in the matching.
  - Looking at 0.2, its only other neighbor, 0.8, has rank greater than it, so 0.2 **is** in the matching.
  - Since 0.2 is in the matching, 0.4 **is not** in the matching.

- Since 0.5 has rank lower than 0.7, and neither 0.3 or 0.4 are in the matching, we know know that 0.5 **is** in the matching.

The question now is: *Is randomness really going to help us?*

**Algorithm 5** Maximal Matching Oralce

```
procedure MAXIMALMATCHINGORACLE(e)
    for all e' ∈ neighbors(e) do
        if rank'_e < rank_e then
            if e' ∈ M then
                Output: 'e ∉ M'
    Output: 'e ∈ M'
```

The correctness of Algorithm 5 follows from the correctness of the greedy algorithm. The $E[\#\textit{oracle calls}] = 2^d$.

**Claim** $E[\#\textit{queries/oracle call}] = 2^{O(d)}$

**Proof** When determining if a node $e$ is in the matching, we can explore the sub-tree of a $d$-ary tree of the neighbors of each edge.

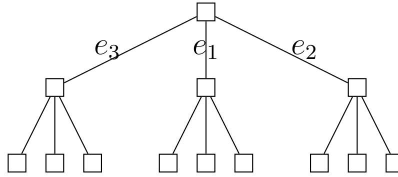


**Figure 7**: d-ary tree of $e$'s neighbors.

Figure 7 is an example neighbor tree for a node $e_0$ where the maximum degree, $d$, is 3. The number of nodes in the tree, is $d^k$ where $k$ is the height of the tree. The number of nodes in the tree is the number of edges we will have to explore to determine if $e_0$ is in the matching.

Now we want to bound the size of the tree. We know that we will only keep paths that are monotonically decreasing in rank, as once a edge has no neighbors with rank less than itself, we can determine whether or not it is in the matching. The probability of a path explored (monotonically decreasing) is $P(\textit{any path } P \textit{ of length } k \textit{ is explored} = \frac{1}{(k+1)!}$. There are at most $d^k$ paths of length $k$. In order to get the total expected number of edges explored, we must sum over all $k$. Therefore, $E[\textit{total \# of edges explored}] = \sum_{k=0}^{\infty} \frac{d^k}{(k+1)!} \leq \frac{e^d}{d}$. ■