0368.416701 Sublinear Time Algorithms 23 November, 2015

Lecture 6

Lecturer: Ronitt Rubinfeld *Scribes: Chen Ziv, Eliav Buchnik, Ophir Arie, Jonathan Gradstein*

Lesson overview

1. Using the oracle reduction framework for approximating the size of maximal matchings (and vertex covers) in a sublinear number of queries.
2. Implementation of the oracle.
3. Approximating the average degree of a graph.

1 Vertex cover approximation using Maximal matching

Continuing our effort from the last lecture to compute the vertex cover of a graph by simulating a distributed algorithm we will try get an approximation using the graph maximal matching.

1.1 Maximal matching

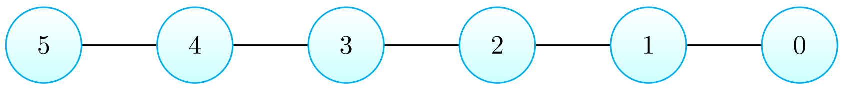
1.1.1 Matching

Matching (M) of a graph $G = (V, E)$ is a set of such that no two edges share a common vertex.

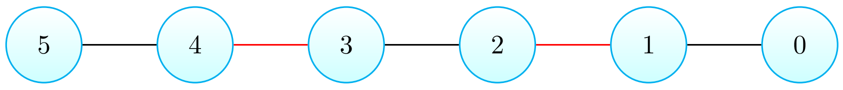
1.1.2 Maximal VS Maximum

Maximal matching (MM) is a matching that cannot be increased without violating the matching.
Maximum matching is a matching with maximum number of edges.

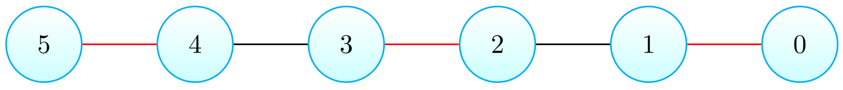
Example consider the following graph



Possible **maximal matching** (red edges)



Possible **maximum matching** (red edges)



1.2 Relation between vertex cover and MM

Reminder: VC is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.
$|VC| \geq |MM|$**:** $\forall (u, v) \in MM$ at least u or v is in the VC (definition of VC). Since the edges in the MM don't share a vertex it is clear that $|VC| \geq |MM|$
$|VC| \leq 2|MM|$**:** We will create a VC from the MM. $\forall (u, v) \in MM$ we will add u, v to the VC. Once we finished

constructing the VC if exists an edge (u', v') that is not covered by the VC then we could have added this edge to the MM without violating the Matching contradicting the assumption of MM. We constructed a VC of size $2MM$ thus the optimal is less than that.

Assuming we can compute the $|MM|$ we get a 2 approximation to the $|VC|$. The algorithm that we will see will find an approximation to the $|MM|$ in sub linear time thus yielding an approximation to the $|VC|$.

1.3 Greedy MM algorithm

```
Data: $G = (V, E)$
Result: $M$
$M \leftarrow \emptyset$;
for $(u, v) \in E$ do
    if $u$ nor $v$ is matched then
        $M \leftarrow M \cup (u, v)$
    end
end
```

Algorithm 1: Maximal matching algorithm

Claim. M is a maximal matching

Proof: Lets assume that exists an edge $e = (u, v)$ such that $e \notin M$ and $\{e\} \cup M$ is a matching. Lets simulate the algorithm up to the point that the algorithm encounters the edge, since $M \cup \{e\}$ is a matching the algorithm will add the edge to M with contradiction to the assumption. ■

1.4 Oracle reduction framework

Assume that we are given an oracle O such that given an edge e the oracle outputs "YES" if $e \in MM$ and "NO" else. Given that we have such an oracle we can build a sub linear algorithm to estimate the $|MM|$.

```
Data: $G = (V, E), O$
Result: estimate of $|MM|$
$S \leftarrow \frac{8}{\epsilon^2}$ nodes sampled iid;
$\forall v \in S$, define $X_v$;
for $v \in S$ do
    if $\exists w \in N(v)$ such that $O((v, w)) =$ "YES" then
        $X_v = 1$
    else
        $X_v = 0$
    end
end
Output $((\frac{\sum_{v \in S} X_v}{|S|})n)\frac{1}{2} + \frac{\epsilon n}{2}$;
```

Algorithm 2: Maximal matching algorithm

Claim. The expected output of the algorithm is $|MM| + \frac{\epsilon n}{2}$

Proof:

$$E[\sum_{v \in S} X_v] = \sum_{v \in S} E[X_v]$$

$$\sum_{v \in S} E[X_v] = |S| E[X_v]$$

We sample iid the nodes from the graph thus the probability of sampling a node in the maximal matching is $\frac{2|MM|}{n}$.

$$|S| E[X_v] = \frac{2|S||MM|}{n}$$

Plunging the result to the output of the algorithm

$$E[((\frac{\sum_{v\in S} X_v}{|S|})n)\frac{1}{2} + \frac{\epsilon n}{2})] = \frac{n}{2|S|}E[\sum_{v\in S} X_v] + \frac{\epsilon n}{2}$$

$$\frac{n}{2|S|}E[\sum_{v\in S} X_v] + \frac{\epsilon n}{2} = \frac{n}{2|S|}\frac{2|S||MM|}{n} + \frac{\epsilon n}{2} = |MM| + \frac{\epsilon n}{2}$$

■

1.4.1 Bounding the error probability

X_v are identically distributed Bernoulli random variables with $p = \frac{2|MM|}{n}$ therefore by Hoeffding's inequality with probability $> 1 - 2e^{-2\epsilon^2|S|}$

$$(p-\epsilon)|S| < \sum_{v\in S} X_v < (p+\epsilon)|S|$$

$$(\frac{2|MM|}{n} - \epsilon)|S| < \sum_{v\in S} X_v < (\frac{2|MM|}{n} + \epsilon)|S|$$

Taking $|S|$ to be $\frac{10}{\epsilon^2}$ yields that with probability $> \frac{2}{3}$

$$|MM| - \frac{\epsilon n}{2} < \frac{n}{2|S|}\sum_{v\in S} X_v < |MM| + \frac{\epsilon n}{2}$$

Since the algorithm adds an additional $\frac{\epsilon n}{2}$ to the output result

$$|MM| < \frac{n}{2|S|}\sum_{v\in S} X_v < |MM| + \epsilon n$$

This completes the error bounding of the algorithm.

2 Implementation of the oracle

Remark. Given an order on the edges of the graph, the greedy sequential algorithm for maximal matching (defined earlier) deterministically outputs a unique maximal matching.

Definition. Given an order on the edges, the *dependency chain* of edge e is the set of all adjacent edges to e that precede it in the order.

Note that in the greedy sequential algorithm, the decision whether to include an edge in the matching or not depends only on the dependency chain of this edge. Hence to implement an oracle O_M we can consider only the dependency chain of the queried edge and check recursively whether the dependency chain edges are in the matching. The problem with this approach is that the dependency chains can be very long:

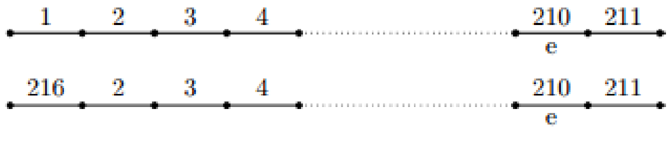


Figure 1: Example for two different rankings that result in long dependency chains. Note that in both cases in order to determine whether the edge marked e belongs to M the algorithm must recursively check all edges preceding it in the ranking

To overcome the difficulty of long dependency chains, we assign a random ordering (ranking) to the edges. Formally, define the ranking as $r\colon E \to [0,1)$. We also assume edges have distinct rankings: $\forall e, e' \in E, r(e) \neq r(e')$.

We'll see later that with this random order the expected size of the dependency chains is not too long.
The implementation of the oracle is as follows:

```
Input  : An edge $e$ and ranking $r\colon E \to [0,1)$
Output: "Yes" if $e \in M$ and "No" otherwise
for every edge $e'$ adjacent to $e$ do
    if $r(e') < r(e)$ then
        recursively run this algorithm on $e'$;
        if received that $e' \in M$ then
            return "No";
        end
    end
end
return "Yes";
```

Algorithm 3: Oracle for determining whether $e \in M$

The correctness of this algorithm follows directly from the correctness of the greedy algorithm.
Figure 2 is an example of an execution of the algorithm.

Claim. The expected number of queries performed by the oracle algorithm is $O(2^{O(d)})$.

Corollary. The total query complexity of the algorithm using this oracle is $O(\frac{2^{O(d)}}{\epsilon^2})$

Proof of claim. For each $e \in E$ consider the tree of queries that are recursively called by the algorithm. The root of this tree is e, and the children of each node in the tree are its adjacent edges (we allow edges to appear several times in the tree). By the algorithm, the paths in the tree that are explored are monotone decreasing in ranking. Given a path of length k (corresponding to $k+1$ edges) the probability that it is monotone decreasing is $\frac{1}{(k+1)!}$. The values of the edges are chosen i.i.d from $r\colon E \to [0,1)$. Therefore, each permutation of the $k+1$ edges occurs with equal probability. Only one of these permutations (r gives distinct values for each edge) is monotone decreasing, which means that:

$$Pr[\text{a path of length k is explored}] = \frac{1}{(k+1)!}$$

Every edge has up to $2d$ neighbors, so the number of nodes in the k-th level of the tree is bounded by $(2d)^k$. Therefore:

$$E[\#\text{ of paths explored at distance k}] \leq \frac{(2d)^k}{(k+1)!}$$

which implies that:

$$E[\text{total \# of edges explored}] \leq \sum_{k=0}^{\infty} \frac{(2d)^k}{(k+1)!} \leq \frac{e^{2d}}{2d}$$

For each node we must check all of its neighbors (of which there may be d), so the expected sample complexity of the oracle algorithm is bounded by:

$$d \cdot E[\text{total \# of edges explored}] = O(e^{2d})$$

which is of course also $O(2^{O(d)})$, as needed. □

Remark. Our algorithm uses $O(n)$ samples in the worst case. We can bound the sample complexity by forcibly cutting off the oracle (and outputting randomly) after it has used $c2^{O(d)}$ samples for some constant c, which by Markov's inequality will ensure that there is a low probability of this happening, and hence a low probability of error.

Remark. There are some recent improvements to this algorithm that have almost linear sample complexity in d.

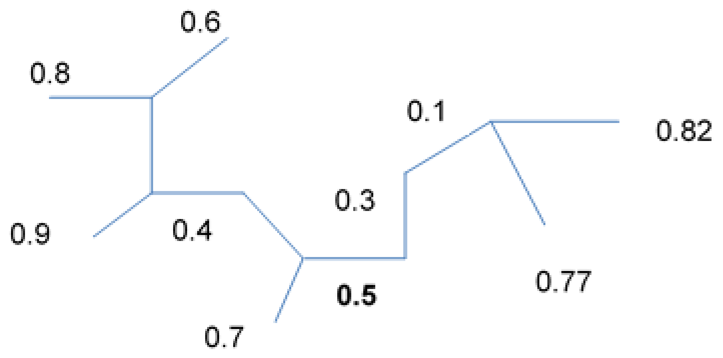


Figure 2: The oracle will determine whether the edge 0.5 (bolded) is part of the maximal matching. We'll consider all its adjacent edges:

1. 0.7 - this edge comes after 0.5, so we'll skip it.
2. 0.3 - we'll check it recursively:
   - 0.1 -
     - 0.82 - this edge comes after 0.1, so we'll skip it.
     - 0.77 - this edge comes after 0.1, so we'll skip it.
     - 0.1 is part of the maximal matching.
   - 0.1 is part of the maximal matching so 0.3 isn't part of the maximal matching.
3. 0.4 - we'll check it recursively:
   - 0.9 - this edge comes after 0.4, so we'll skip it.
   - 0.2 - we'll check it recursively:
     - 0.6 - this edge comes after 0.2, so we'll skip it.
     - 0.8 - this edge comes after 0.2, so we'll skip it.
     - 0.2 is part of the maximal matching.
   - 0.2 is part of the maximal matching so 0.4 isn't part of the maximal matching.
4. All the adjacent edges of 0.5 aren't part of the maximal matching, therefore 0.5 is part of the maximal matching.

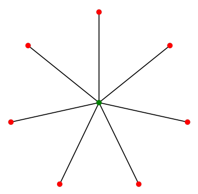
3 Approximating average degree

For a given graph G, we would like to approximate the *average degree*, which is defined to be $\bar{d} = \frac{\sum_{v \in V} d(v)}{n}$. We will make several assumptions:

1. G is a simple graph: it does not contain multiple edges or loops.
2. G is not "ultraspare", meaning: $|E| = m = \Omega(n)$.
3. Our model is slightly different to the graph model seen before, which will allow us to make stronger queries:
   (a) **Degree queries:** return $d(v)$ for input vertex v.
   (b) **Neighbor queries:** for input vertex v and index j, return the j-th neighbor of v.

An attempt at a naive solution

Randomly choose $v_1, ..., v_s$ vertices, and output $\frac{1}{s}\sum_{i=1}^{s} d(v_i)$.
The problem with this approach is that for certain input graphs the variance can be very high.
Consider for example the star graph on n vertices, where one vertex has $n-1$ neighbors and all the rest have one neighbor. The average degree is $2 - \frac{2}{n}$ which approaches 2.



However, any sample that we choose that doesn't happen to contain the center will return an estimate of 1, which is a bad multiplicative error.
This allow us to conclude that we must make $\Omega(n)$ queries for this approach to be effective.
In the next lecture, we'll see a sublinear algorithm for approximating the average degree.