COMS E6998: Algorithms for Massive Data (Fall'23)	Nov 8, 2023

Lecture 18: Sublinear Time Algorithm for Vertex Cover Estimation

Instructor: *Alex Andoni* Scribes: *Walter McKelvie*

Definition 1. *Consider a graph G. V is a vertex cover for G if for all edges in G are incident to at least one $v \in V$.*

Our goal is to compute, for given G, the minimum vertex cover

$$\text{MVC} := \min_{V \text{ vertex cover for } G} |V|$$

Unfortunately this is NP-hard to compute exactly. So, we instead must settle for an approximation algorithm.

Definition 2. *M is maximal matching of G if*

1. *M is a matching of G*
2. *Cannot add another edge to M and still have a matching.*

We note that a maximum matching (i.e., a matching with the greatest number of edges) is maximal, but a maximal matching not necessarily a maximum matching. The problems of estimating the size of a maximal matching and a MVC are related by the following theorem.

Theorem 3 (Garril-Yannakakis)**.** *Let M be a maximal matching. Then $|M| \leq MVC \leq 2 \cdot |M|$.*

Proof. To prove that $\text{MVC} \leq 2 \cdot |M|$, we let V be endpoints of M. $|V| = 2|M|$, so we see $|V| = 2|M|$. Next, to show that $|M| \leq \text{MVC}$, we let V be a minimum vertex cover. $|V| \geq |M|$ because for all edges in M at least one endpoint is in V. □

This shows that we can approximate MVC to a factor of 2 if we can estimate the size of a maximal matching, which will motivate our approximation algorithm.

Algo: Compute a maximal match M by greedily adding edges.

Theorem 4 (Nguyen-Onak 08)**.** *There is an algorithm which outputs an approximation $\widehat{MVC}$ such that*

$$MVC - \epsilon n \leq \widehat{MVC} \leq 2 \cdot MVC + \epsilon n$$

in time

$$O\left(\frac{d}{\epsilon^2} \cdot e^d\right)$$

where d is max degree of G.

Note: Same factor for 2 approximation as above. Does not depend on the number of nodes, only the max degree–this makes it good for sparse graphs.

Pf:

Idea: estimate size of a maximal matching M.

Observation 5. *Finding maximal matching in G is the same as finding maximal independent set (IS) in G', where G' is the graph where the nodes are edges of G, and where two vertices in G' are connected if their corresponding edges in G share a vertex. This follows from the definitions; a maximal matching of G is a set of edges which don't share a vertex, which is exactly a set of vertices in G' which don't share an edge.*

So, it would suffice to estimate size of some maximal independent set in G'. This would motivate the following (naive) algorithm:

Algorithm 6. *GY Algorithm*

1. $I \leftarrow \emptyset$
2. *For $i \in$ vertices of G' in arbitrary order:*
    1. *If i not adjacent to some $j \in I$*
        i. $I := I \cup \{i\}$
3. *Return* $2 \cdot |I|$

Unfortunately, this does not give sublinear time. Actually generating an independent set I requires that the complexity grows with m, which we don't want. We will need to do this without actually computing I.

Idea: Fix some (distribution over, independent of samples) I which is implicit. Sample a bunch of nodes in G' and check if $i \in I$ without ever computing I. Then, our estimate is

$$\hat{I} = \frac{n}{k} \cdot \sum_{j=1}^{k} \mathbb{1}_{\{i_j \in I\}}$$

This requires a so-called Local Oracle: We have a graph G'. We want our Local Oracle to be able to tell us whether any particular node i is in a maximal independent set I, which we can choose randomly. In fact, we can design such an oracle, where query time is $O(d \cdot e^d)$. Let I be GY algorithm's output with random order. To implement this random order, pick $r_i \in [0, 1]$ for all nodes i. Order i in increasing order of r_i's.

First, we define a helper function

Algorithm 7. *Get-r*(i)

1. *If r_i is not yet chosen:*
    1. *Choose r_i uniformly from* $[0, 1]$.
    2. *Store in a hash table* (i, r_i).
2. *Return r_i from the hash table.*

and our final local oracle is defined by

Algorithm 8. *Oracle*(i, G')*:*

1. *For $j \in N(i)$:*

   1. *If Get-r(j) $<$ Get-r(i):*

      i. *If Oracle(j) then return no.*

2. *Return yes.*

This asks "are all of my neighbors who the algorithm will try to add to I before me, not in I." Since these are the only possible reasons that i will not be added to I, correctness is clear. Less clear is the runtime.

Theorem 9. *The runtime of the oracle is $O(e^d)$ in expectation.*

Proof. The "bad case" here is that there is a chain of decreasing r_i which causes the oracle to recursively explore the entire graph; we must claim that is unlikely. In particular,

$$\mathbb{E}(\text{\# visited nodes in an oracle query}) \leq e^d$$

We can show the above as follows. Fix an arbitrary path P of length k. Then

$$\mathbb{P}(\text{follow path } P) = \frac{1}{(k+1)!}$$

This means that the expected number of visited nodes is

$$\mathbb{E}(\text{\#visited nodes}) \leq \sum_{k \geq 1} (\text{\#path of length } k) \cdot \mathbb{P}(\text{follow path of length } k)$$

$$\leq \sum_{k \geq 1} d^k \cdot \frac{1}{(k+1)!} \leq \frac{1}{d} \sum_{k \geq 0} \frac{d^{k+1}}{(k+1)!} = \frac{e^d}{d}$$

using the Taylor expansion for e^d.

Note that once we visit a node, we spend $O(d)$ time to check its neighborhood, hence $O(e^d)$ total time. □

Let's compute the total runtime. We know the number of nodes in $G' \leq dn$, and we aim for an additive error of $\epsilon n = \frac{\epsilon}{d} \cdot n$. Hence, it suffices to sample $k = \frac{d^2}{\epsilon^2}$ nodes in G'. We can shave off a factor of d by remembering that we are estimating the size of the independent set I in G', which is equivalent to some matching M in G. We can think of the above algorithm is directly computing the size of M by sampling nodes and checking whether they are matched. In particular, for additive ϵn nodes (to the size of M), it is enough to sample $k = O(1/\epsilon^2)$ nodes and check whether each is in the matching M. For each such node, we check whether any of its incident edges participates in M (equivalently I in G'). The total runtime becomes

$$O(k \cdot d \cdot e^d) = O\left(\frac{d}{\epsilon^2} \cdot e^d\right).$$

There are several improvements to the above algorithm, stated below.

Theorem 10 (Yoshita, Yamamoto, Ito 09)**.** *The above algorithm can be improved to poly(d/ϵ) runtime. This is achieved through an extra heuristic: in the Oracle loop, go in order of increasing r_j's. Through* *probability wrangling, the expected number of recursive calls (over input i and the ordering r) comes down to $1 + m/n \leq 1 + d$ which yields the desired result.*

Theorem 11 (Onak, Ron, Rosen, Rubinfeld 12)**.** *This problem can be solved with runtime quasilinear in d:*

$$O\left(\frac{d}{\epsilon^3}\log^2\frac{d}{\epsilon}\right)$$

In fact, this is the best known result.