CS 590 Sublinear algorithms March, 2015

Lecture 15 & 16

Lecturer: Elena Grigorescu *Scribe: Wuwei Zhang, Young-San Lin, Xiangyu Bu*

1 Introduction

Finding the minimum vertex cover of a graph is a well-known NP-hard problem. Researchers have been designing algorithms to approximate the minimum vertex cover of a graph in more accurate and efficient ways. In this lecture note, we summarize and discuss sublinear-time algorithms for approximating the *size* of the minimum vertex cover.

To describe how accurately an algorithm approximates the minimum vertex cover, we define the following notation, which gives the multiplicative and additive error:

Definition 1 ($(\alpha$-$\epsilon)$-approximation) *Let the optimal solution be OPT. An $(\alpha, \epsilon)-$approximation solution SOL approximates OPT so that*

$$OPT \leq SOL \leq \alpha \cdot OPT + \epsilon \cdot n$$

where $\alpha \geq 1$, $\epsilon \in [0, 1)$, and n is the size of the input.

To describe the running time of an algorithm, we also give the following notation:

Definition 2 (Soft-O $\tilde{O}$) *$f(n) = O(\tilde{g(n)})$ is a short hand of $f(n) = O(g(n) \log^k g(n))$ for some constant k.*

1.1 Preliminary Work

Several approaches have been researched to approximate the size of minimum vertex cover (minVC) of a graph. The most notable ones are to estimate minVC

1. by maximal matching,
2. by calculating minVC in locally distributed ways,
3. by using a streaming model.

Our survey discusses the first two approaches but focuses on the first approach only.

1.1.1 Matching and Vertex Cover

A series of algorithms to approximate the size of minimum vertex cover of a graph were derived from the algorithm to produce a maximal matching of the graph. We first define matching and maximal matching, and discuss how a maximal matching can be used to approximate the minimum vertex cover.

Definition 3 (Matching) *Let G = (V, E) be a graph. A matching M in G is a set of edges such that no two edges in M share a common vertex.*

Definition 4 (Maximal Matching) *Let M be a matching of $G = (V, E)$. M is a maximal matching of G iff. $\forall\ e \in E\text{-}M$, $M \cup \{e\}$ is not a matching.*

The set of endpoints of edges in a maximal matching forms a minimum vertex cover of a graph, because from the definition of maximal matching, all the edges in the graph have at least one endpoint included in the set. In other words, the set covers all edges. The set of endpoints in a maximal matching results in a 2-approximation of the size of minimum vertex cover.

Theorem 5 (Bounds for min vertex cover) *Let M be the size of any maximal matching for G, then the size of a minimum vertex cover is lower-bounded M and upper-bounded by $2M$.*

A well-know algorithm that produces *one* maximal matching of a graph is proposed by Gavril and Yanakakis [PS98]. It is a greedy algorithm that runs in time linear in the size of the graph.

Algorithm 1 Greedy Sequential Matching

Input: $G(V, E)$
Output: C

```
1: C = ∅
2: for each e = (u, v) ∈ E do
3:    if neither u nor v is used by any edge in C then
4:       C = C ∪ {u, v}
5:    end if
6: end for
```

So far, all sublinear time algorithms that pursue the maximal matching approach are derived from the Greedy Sequential Algorithm.

1.1.2 Distributed Approach

Another approach to estimate the size of minimum vertex cover is to estimate it on part of a graph and gradually expand the result and clear the residual graph.

In 2007, Michael Parnas and Dana Ron [PR07] showed how to obtain a $(2, \epsilon)$-estimate of minimum vertex cover in time $d^{O(\log \frac{d}{\epsilon^3})}$, where d is the maximal degree of a vertex in the graph. The algorithm uses locally distribution approach assumes synchronous network, and ignores all vertices whose degree is 1. It preferably picks vertices with higher degrees and keeps distributed sites in sync by shrinking and updating the residual graph. The sublinear time version is made by adding a reduction algorithm (refer to [PR07]) that adaptively forms a subgraph of original graph as input for the local distributed algorithm.

Parnas and Ron proved that at least a **linear** dependence on the average degree is necessary, meaning that $\Omega(\bar{d})$ queries will be needed to obtain an (α, ϵ)-estimate. Note that $d = \frac{\bar{d}}{\epsilon}$, where $\bar{d}$ is the average degree for vertices in the graph.

Algorithm 2 Distributed Matching

Input: Graph $G = (V, E)$, d
Output: C

1: $C = \emptyset$
2: **for** $i = 1$ *to* $log(d)$ **do**
3: $S = \{v \in \mathrm{E} \mid deg(v) \geq \frac{d}{2^i}\}$
4: Remove all edges incident to vertices in S
5: Update degrees of the residual graph
6: $C = C \cup S$
7: **end for**

The run time had been significantly improved in a sequence of papers [MR09, NO08, YYI09], where the best result due to Yoshida, Yamamoto, and Ito [YYI09] is an upper bound of $O(\frac{d^4}{\epsilon^2})$ and it can be easily adapted to given an upper bound of $O(\frac{\bar{d}}{\epsilon^4})$ for graphs with bounded average degree $\bar{d}$.

1.1.3 A Lower Bound for Queries

Parnas and Ron [PR07] showed that obtaining an (α, ϵ)-estimate for any $\alpha \geq 1$ and $\epsilon < \frac{1}{4}$ requires $\Omega(\bar{d})$ queries if $\bar{d} = O(\frac{n}{\alpha})$. Also, Trevisan mentioned that for any constant $\gamma > 0$, $\Omega(\sqrt{n})$ queries are needed to obtain any $(2 - \gamma, \epsilon)$-estimate.

1.2 Nearly Optimal Sublinear Approach

Apparently sublinear-time algorithms cannot read the whole graph. Therefore, sublinear versions (discussed in next sections) of Greedy Sequential Matching algorithm [PS98] and Local Distributed Matching algorithm [PR07] agree on three types of queries to the target graph:

1. Query the degree $\deg(v)$ of any vertex v.
2. Query the ith neighbor of any vertex v (the order is algorithm-dependent).
3. Query if there exists an edge between two vertices u and v.

Based on the three basic types of queries, two common query models are formed:

1. Query on vertex v : get the degree of v, can also query i^{th} neighbor of vertex v.
2. Query on edges: beside the above mentioned, check whether there exists and edge $e = (u, v)$. (Graph representation is adjacency matrix)

Based on the two approaches and three basic queries, researchers Onak, Ron, Rosen, and Rubinfeld summarized the best work of their predecessors, and proposed an algorithm that give a (2-ϵ)-estimate of the size of minimum vertex cover $VC_{opt}(G)$ with running time $\tilde{O}(d)\cdot\mathrm{poly}(\frac{1}{\epsilon})$ [ORRR12]. In general, the algorithms approximate the size by producing (an estimate of) maximal matching, and speed up with an idea similar to the reduction algorithm in local distributed approach. The nearly optimal version focuses on the first model, but authors have proved that the algorithm can be modified to work in the second model as well.

2 Approx-VC-I Algorithm

The Approx-VC-I algorithm described in [ORRR12] is a summary of the consecutive work by [PR07], [NO08], [MR09], and [YYI09]. It is directly derived from Greedy Sequential Matching algorithm.

Approx-VC-I [PR07] uniformly, independently, and randomly selects $\Theta(\frac{d^2}{\epsilon^2})$ edges. For each edge selected, it calls a **Maximal Matching Oracle**. This oracle takes an edge e as input and answers whether e is in the maximal matching M or not. Then based on how many edges get "accepted" by the oracle, the algorithm reports an estimate of the size of the maximal matching, which is then used to calculate the size of minimum vertex cover. The number of sampled edges ensures that with high constant probability, the additive error estimate is $O(\frac{m\epsilon}{d}) \leq \epsilon \cdot n$

Algorithm 3 Approx-VC-I

Input: $G(V, E)$
Output: *counter*

```
1: Randomly pick Θ(d²/ε²) edges from E.
2: counter ← 0
3: for each e we picked do
4:    Max-Matching-Oracle(e)
5:    if Accept then
6:       counter ← counter + 1
7:    end if
8: end for
```

2.1 Max-Matching Oracle

In order to understand the correctness of this algorithm and achieve a sublinear run time, it is critical to understand how the Max-Matching Oracle works. The idea is to assume that edges in G have a fixed ranking (or ordering) which uniquely determines a maximal matching as follows:

1. Proceeding along the edges based on ranking.
2. Add e to maximum matching M that does not share an end point witch any edge already in M.

The ranking can be chosen independently and randomly at the beginning. With the ranking determined, the oracle simulates the above procedure and creates a partial ordering. To determine whether an edge e is in M or not, we simply need to check whether adjacent edges of e with lower rank are in M or not. (If so, return true.)

2.2 Efficiency

The Approx-VC-I algorithm has a query complexity and run time of $O(\frac{d^4}{\epsilon^2})$ proved in [YYI09].

3 Approx-VC-II Algorithm

3.1 Overview

Approx-VC-II is the algorithm discovered in [ORRR12] by Onak, Ron, Rosen, and Rubinfeld. It has a linear dependence on the maximum degree d and it is obtained by modifying Approx-VC-I as the following: Instead of sampling edges, sample vertices and utilize a vertex cover oracle to determine the size of the minimum vertex cover.

3.2 Vertex Cover Oracle

The vertex cover oracle takes a vertex v as input and output whether v is in the minimum vertex cover by calling the Max-Matching Oracle mentioned in Approx-VC-I.
To achieve the same level of accuracy as Approx-VC-I, sampling $\Theta(\frac{1}{\epsilon^2})$ vertices is sufficient.

Theorem 6 *For a graph $G = (V, E)$ and any fixed choice of ranking π, let $C = C^{\pi}(G)$. Suppose that we uniformly and independently select $s = \Theta(\frac{1}{\epsilon^2})$ vertices v from V. Let t be a random variable equal to the number of selected vertices that belong to C.*

$$|C| - \epsilon \cdot n \leq \frac{t}{s} n \leq |C| + \epsilon \cdot n$$

Algorithm 4 Approx-VC-II

Input: $G(V, E)$
Output: *counter*
1: Randomly select$\Theta(\frac{1}{\epsilon^2})$ vertices from V.
2: $counter \leftarrow 0$
3: **for** each v we picked **do**
4: **if** $VO^{\pi}(v)$ **then**
5: $counter \leftarrow counter + 1$
6: **end if**
7: **end for**

Algorithm 5 Vertex Oracle $VO^{\pi}(v)$

Input: v
Output: Decide whether v is in the maximum matching based on π.
1: Let $e_1, ..., e_t$ be edges incident to vertex v in increasing rank.
2: **for** $i = 1$ to t **do**
3: **if** $MO^{\pi}(e_i)$ =TRUE **then**
4: **return** TRUE
5: **end if**
6: **end for**
7: **return** FALSE

4 Bounding the Excepted Number of Calls to Oracle

Let $N(\pi, v)$ denote the number of different edges such that a call given each such edge e $MO^{\pi}(e)$ was made to the Max-Matching Oracle in the course of the computation of $VO^{\pi}(v)$.

Algorithm 6 Edge Oracle $MO^{\pi}(e)$

Input: e
Output: Decide whether e is in the maximum matching based on π.

```
if MO^π(e) already compted then
    return answer
end if
Let e_1, ..., e_t be edges that share one vertex with e in increasing rank.
i ← 1
while π(e_i) < π(e) do
    if MO^π(e_i) =TRUE then
        return FALSE
    else
        i ← i + 1
    end if
end while
return TRUE
```

Theorem 7 *Let G be a graph with m edges and average degree $\bar{d}$, let $\rho = \frac{d_{max}}{d_{min}}$. The average value of $N(\pi, v)$ taken over all ranking π and vertices is $O(\rho \cdot \bar{d})$.*

$$\frac{1}{m!}\frac{1}{n}\sum_{\pi\in\Pi}\sum_{v\in V} N(\pi, v) = O(\rho \cdot \bar{d})$$

Let $X^{\pi}(v, e) = 1$ if e is visited in the course of execution of $VO^{\pi}(v, e)$. If e and e' share v and $\pi(e') < \pi(e)$, then $MO^{\pi}(e) = FALSE$.

$$N(\pi, v) = \sum_{e\in E} X^{\pi}(v, e)$$

Let $X_k(e) = \sum_{\pi\in\Pi}(X^{\pi}(v_a(\pi_k), e) + X^{\pi}(v_b(\pi_k), e))$ where π_k is the edge or rank k and $v_a(\pi_k)$ and $v_b(\pi_k)$ is its two endpoints. That is , $X_k(e)$ is the total number of calls made to the maximal matching oracle on e when summing over all rankings π. Notice that

$$\sum_{k=1}^{m} X_k(e) = \sum_{\pi\in\Pi}\sum_{v\in V} deg(v) \cdot X^{\pi}(v, e)$$

Lemma 8 *For every edge e and $k \in [m-1]$, $X_k(e) \leq 2(m-1)! + (k-1)(m-2)! \cdot d$*

Proof We skip the details but outline the main idea for this proof. For details, see [ORRR12].

If we fix k and e, for a ranking π, let π' be another ranking such that $\pi'_{k+1} = \pi_k$, $\pi'_k = \pi_{k+1}$, and $\pi'_i = \pi_i$ if $i \neq k$ and $i \neq k+1$.

For each edge e where $\pi(e) < k$, we have $MO^{\pi}(e) = MO^{\pi'}(e)$ because $\pi(e) = \pi'(e)$.
Let Π_k denotes those rankings π in which π_k and π_{k+1} share an endpoint. Thus, if $\pi \in \Pi_k$, then $\pi' \in \Pi_k$ because of the definition. For two edges $e' = (v_1, v_2)$ and $e'' = (v_2, v_3)$: let $v_c(e', e'') = v_c(e'', e') = v_2$, $v_d(e', e'') = v_1$, $v_d(e'', e') = v_3$. If e' and e'' are parallel, then $v_d(e', e'') = v(e'', e') = v_1, v_c(e', e'') = v_c(e'', e') = v_2$. If e' is a loop, then $v_d(e'', e'') = v_c(e'', e'') = v_1$

By some great properties base on π and π',

$$X_{k+1}(e) - X_k(e) = \sum_{\pi \in \Pi_k} X^{\pi'}(v_d(\pi'_{k+1}, \pi'_k), e) - \sum_{\pi \in \Pi_k} X^{\pi}(v_d(\pi_{k+1}, \pi_k), e)$$

Consider all the cases of how v_1, v_2, and v_3 are connected, it turns out that only the following two cases make $X^{\pi'}(v_d(\pi'_{k+1}, \pi'_k), e) = 1$ and $X^{\pi}(v_d(\pi_{k+1}, \pi_k), e) = 0$:

1. $v_1 \neq v_2$, $v_1 \neq v_2$, $v_2 \neq v_3$, $e' = (v_1, v_2)$ with rank $\pi'_{k+1} = \pi_k$, and $e'' = (v_2, v_3)$ with rank $\pi_{k+1} = \pi'_k$.
2. $v_1 = v_2$, $v_2 \neq v_3$, $e' = (v_1, v_2)$ with rank $\pi'_{k+1} = \pi_k$, and $e'' = (v_2, v_3)$ with rank $\pi_{k+1} = \pi'_k$.

Let $\Pi_k^{e,1}$ denote the set of all rankings $\pi' \in \Pi_k$ where $e = e''$ and $X^{\pi'}(v_d(\pi'_{k+1}, \pi'_k), e) = 1$, assuming the worst case that $X^{\pi}(v_d(\pi_{k+1}, \pi_k), e) = 0$.

Let $\Pi_k^{\neg e}$ denote the set of all rankings $\pi' \in \Pi_k$ where $e \neq e'$, $e \neq e''$ and $X^{\pi'}(v_d(\pi'_{k+1}, \pi'_k), e) = 1$ while $X^{\pi}(v_d(\pi_{k+1}, \pi_k), e) = 0$.

Let $\Pi_k^{e,0}$ denote the set of all rankings $\pi' \in \Pi_k$ where $e = e''$ and $X^{\pi}(v_d(\pi_{k+1}, \pi_k), e) = 0$.

By the analysis through the two above mentioned cases that make $X^{\pi'}(v_d(\pi'_{k+1}, \pi'_k), e) = 1$ and $X^{\pi}(v_d(\pi_{k+1}, \pi_k), e) = 0$, we observe that there exists an injection from $\Pi_k^{\neg e}$ to $\Pi_k^{e,0}$.

As a result, $X_{k+1}(e) - X_k(e) \leq |\Pi_k^{e,1}| + |\Pi_k^{\neg e}| \leq |\Pi_k^{e,1}| + |\Pi_k^{e,0}| \leq (m-2)! \cdot d$ because each ranking $\pi' \in \Pi_k^{e,1} \cup \Pi_k^{e,0}$ is decided by first setting the rank of e to be k, then select another edge incident to v_2 of e which have at most $d-1$ choices, and finally one out of the possible $(m-2)!$ rankings.

Eventually, we construct the induction proof. If $e = \pi_1$, then $X^{\pi}(v_a(\pi_k), e) = X^{\pi}(v_b(\pi_k), e) = 1$, and otherwise $X^{\pi}(v_a(\pi_k), e) = X^{\pi}(v_b(\pi_k), e) = 0$. For any fixed e, the number of ranking π such that $e = \pi_1$ is $(m-1)!$, so $X_1(e) = 2(m-1)! \leq 2(m-1)! + (1-1)(m-2)! \cdot d$.

For the induction step, suppose the induction hypothesis holds for $k - 1 \geq 1$, then:

$$\begin{aligned} X_k(e) &\leq X_{k-1}(e) + (m-2)! \cdot d \\ &\leq 2(m-1)! + (k-2)(m-2)! \cdot d + (m-2)! \cdot d \\ &= 2(m-1)! + (k-1)(m-2)! \cdot d \end{aligned}$$

■

Now we can prove **Theorem 7** by **Lemma 8**:

Proof

$$\begin{aligned} \frac{1}{m!}\frac{1}{n}\sum_{\pi \in \Pi}\sum_{v \in V} N(\pi, v) &\leq \frac{1}{m!}\frac{1}{n}\frac{1}{2d_{min}}\sum_{e \in E}\sum_{k=1}^{m} X_k(e) \\ &\leq \frac{1}{m!}\frac{1}{n}\frac{1}{2d_{min}} m[m \cdot 2(m-1)! + \frac{m(m-1)}{2}(m-2)! \cdot d] \\ &= O(\frac{m}{n}\frac{d}{d_{min}}) = O(\rho \cdot \bar{d}) \end{aligned}$$

■

5 Limiting the Exploration of Neighbor Sets

The trick to make the algorithm runs in sublinear time is to limit the exploration of neighbors of sampled vertices. This section explains the high level idea of how to do this.

In order to achieve such speed up (by a factor of ϵ), the algorithm VC-II is modified by the following:

- Replace random ranking with random numbers on (0,1]
- Data structure for accessing neighbors

5.1 Replace Random Ranking With Random Numbers

If we assign edges with random numbers during the execution of the algorithm, it is very unlikely two edge will get the same number since interval (0,1], the probability of two edges getting two identical value, is $\frac{1}{\infty}\frac{1}{\infty} \approx 0$. Thus, it is feasible to assign the "ranking" on the fly.

5.2 Data Structure for Accessing Neighbors

For each vertex v in V, we have a copy of neightbor[v], where neighbor[v] stores edges adjacent to v.
Operations:

- lowest(k): Return an edge(w,r), (w,r) is the edge with k smallest random number in neighbor of v
  - Consider interval $(2^{-i}, 2^{-i+1}]$ and $(0, 2^{-d_*}]$.
  - Decide which edge should be assigned from $(2^{-i}, 2^{-i+1}]$. If needed, assign based on their labels
    - For every edge e =(v,w) in this stage, get w by a neighbor query. Assign a value to e from $(2^{-i}, 2^{-i+1}]$
    - Notify w(update neighbor(w))
    - When a vertex is first called, the interval is $(0, 2^{-d_*}]$.
  - Open a new interval if there already k edges of v get ranked

Attributes

- lb, next_lb for interval. For each call, we set lb = next_lb and next_lb *=2
- dictionary: quickly check assigned edges

When assigning values, we flip a coin with bias $\frac{next\_lb - lb}{1-lb}$

5.3 Query Complexity & Run time

Le the upper bound for expected number of calls to oracles using random numbers be t, w.h.p., this algorithm is $O(t \log^2 dt)$.
Idea: The idealized scenario for every vertex v, the number of edges incident to v are assigned a number is exactly the expected assume no conflicting. Each interval is double the size of previous interval, thus the number of queries is at most 2 * number of calls to MO^{δ}.

Theorem 9 *Let t be an upper bound for the number of MO^{π} get called by algorithm$^{\pi}$, and t fits into a constant number of machine words. Replace π by random numbers, with probability $\frac{4}{5}$, all the following are true:*

- *number of queries is $O(t \log^2(dt))$*
- *total time to compute the answer based on queries to the oracle is $O(t \log^3(dt))$*
- *Let D be the distribution of the answer that oracle gives, $\exists$ D', s.t. convex combination $\frac{4}{5}D + \frac{1}{5}D'$ is the distribution of the oracle described by π*

Since t = $\frac{\rho\bar{d}}{\epsilon^2}$, we have run time O($\frac{\rho\bar{d}}{\epsilon^2} \log^3(\frac{d}{\epsilon})$

6 Towards Optimality

We skip the details of using Chernoff bound to bound the maximum and average degrees. From the previous sections, there exist algorithms that run in O($\frac{d}{\epsilon^3} \cdot \log^3(\frac{d}{\epsilon})$) and O($\frac{\bar{d}}{\epsilon^4} \cdot \log^2(\frac{\bar{d}}{\epsilon})$) and with probability $\frac{2}{3}$, output a $(2, \epsilon n)$-estimation to the minimum vertex cover size.

7 Know Results

7.1 Minimum Vertex Cover

Lower bound:
Parnas and Ron [PR07] showed that obtaining an (α, ϵ)-estimate for any $\alpha \geq 1$ and $\epsilon < \frac{1}{4}$ requires $\Omega(\bar{d})$ queries if $\bar{d} = O(\frac{n}{\alpha})$. Also, Trevisan mentioned that for any constant $\gamma > 0$, $\Omega(\sqrt{n})$ queries are needed to obtain any $(2 - \gamma, \epsilon)$-estimate.

Approximation factor:

- Unless P = NP, it is impossible to achieve a 1.3606-approximation.
- If unique game conjecture is true, then minimum vertex cover cannot be approximated within a factor of 2.

7.2 Minimum Spanning Tree

Lower Bound:
According to Chazelle, Rubinfeld, and Trevisan[CRT05], the lower bound for approximating the weight of MST is $\Omega(dw\epsilon^{-2})$

Best Known Algorithm: $O(dw\epsilon^{-2}\log\frac{dw}{\epsilon})$ with relative error at most ϵ[CRT05].

7.3 Counting Triangles

Lower bound:
With only the degree and neighbor queries, it is proved [GRS11] that a sublinear time algorithm for counting triangles in a graph does not exist. According to a recent paper [ELR15], given the additional vertex-pair query, $\Omega(\min\{m, \frac{m^{\frac{3}{2}}}{\Delta(G)}\})$ queries are necessary where $\Delta(G)$ is the number of triangles in graph G.

Best Known Algorithm: Given an approximation parameter $0 < \epsilon < 1$ and graph G, we want to output an estimate $\hat{\Delta}$ with high constant probablity where $(1-\epsilon)\Delta(G) \leq \hat{\Delta} \leq (1-\epsilon)\Delta(G)$. The expected query complexity is $O(\frac{n}{\Delta(G)^{\frac{1}{3}}} + \min\{m, \frac{m^{\frac{3}{2}}}{\Delta(G)}\}) \cdot \text{poly}(\log n, \frac{1}{\epsilon})$ and the expected running time is $O(\frac{n}{\Delta(G)^{\frac{1}{3}}} + \frac{m^{\frac{3}{2}}}{\Delta(G)}) \cdot \text{poly}(\log n, \frac{1}{\epsilon})$.

References

[CRT05] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM Journal on computing*, 34(6):1370–1379, 2005.

[ELR15] Talya Eden, Amit Levi, and Dana Ron. Approximately counting triangles in sublinear time. *arXiv preprint arXiv:1504.00954*, 2015.

[GRS11] Mira Gonen, Dana Ron, and Yuval Shavitt. Counting stars and other small subgraphs in sublinear-time. *SIAM Journal on Discrete Mathematics*, 25(3):1365–1411, 2011.

[MR09] Sharon Marko and Dana Ron. Approximating the distance to properties in bounded-degree and general sparse graphs. *ACM Transactions on Algorithms (TALG)*, 5(2):22, 2009.

[NO08] Huy N Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 327–336. IEEE, 2008.

[ORRR12] Krzysztof Onak, Dana Ron, Michal Rosen, and Ronitt Rubinfeld. A near-optimal sublinear-time algorithm for approximating the minimum vertex cover size. In *Pro-*

ceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms, pages 1123–1131. SIAM, 2012.

[PR07] Michal Parnas and Dana Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoretical Computer Science*, 381(1):183–196, 2007.

[PS98] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[YYI09] Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. An improved constant-time approximation algorithm for maximum. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 225–234. ACM, 2009.