A Near-Optimal Sublinear-Time Algorithm for Approximating the Minimum Vertex Cover Size

Krzysztof Onak* Dana Ron † Michal Rosen ‡ Ronitt Rubinfeld§

October 5, 2011

Abstract

We give a nearly optimal sublinear-time algorithm for approximating the size of a minimum vertex cover in a graph G. The algorithm may query the degree $\deg(v)$ of any vertex v of its choice, and for each $1 \leq i \leq \deg(v)$, it may ask for the i^{th} neighbor of v. Letting $\mathrm{VC}_{\mathrm{opt}}(G)$ denote the minimum size of vertex cover in G, the algorithm outputs, with high constant success probability, an estimate $\widehat{\mathrm{VC}}(G)$ such that $\mathrm{VC}_{\mathrm{opt}}(G) \leq \widehat{\mathrm{VC}}(G) \leq 2\mathrm{VC}_{\mathrm{opt}}(G) + \epsilon n$, where ϵ is a given additive approximation parameter. We refer to such an estimate as a $(2, \epsilon)$-estimate. The query complexity and running time of the algorithm are $\tilde{O}(\bar{d} \cdot \mathrm{poly}(1/\epsilon))$, where $\bar{d}$ denotes the average vertex degree in the graph. The best previously known sublinear algorithm, of Yoshida et al. (*STOC 2009*), has query complexity and running time $O(d^4/\epsilon^2)$, where d is the maximum degree in the graph. Given the lower bound of $\Omega(\bar{d})$ (for constant ϵ) for obtaining such an estimate (with any constant multiplicative factor) due to Parnas and Ron (*TCS 2007*), our result is nearly optimal.

In the case that the graph is dense, that is, the number of edges is $\Theta(n^2)$, we consider another model, in which the algorithm may ask, for any pair of vertices u and v, whether there is an edge between u and v. We show how to adapt the algorithm that uses neighbor queries to this model and obtain an algorithm that outputs a $(2, \epsilon)$-estimate of the size of a minimum vertex cover whose query complexity and running time are $\tilde{O}(n) \cdot \mathrm{poly}(1/\epsilon)$.

*School of Computer Science, Carnegie Mellon University, E-mail: konak@cs.cmu.edu. Research supported by a Simons Postdoctoral Fellowship and the NSF grant 0728645.

†School of Electrical Engineering, Tel Aviv University. E-mail: danar@eng.tau.ac.il. Research supported by the Israel Science Foundation grant number 246/08.

‡Blavatnik School of Computer Science, Tel Aviv University, E-mail: micalros@post.tau.ac.il

§CSAIL, MIT, Cambridge MA 02139 and the Blavatnik School of Computer Science, Tel Aviv University. E-mail: ronitt@csail.mit.edu. Research supported by NSF awards CCF-1065125 and CCF-0728645, Marie Curie Reintegration grant PIRG03-GA-2008-231077 and the Israel Science Foundation grant nos. 1147/09 and 1675/09.

1 Introduction

Computing the size of a minimum vertex cover in a graph is a classic NP-hard problem. However, one can approximate the optimal value of the solution to within a multiplicative factor of two, via a neat and simple algorithm whose running time is linear in the size of the graph (this algorithm was independently discovered by Gavril and Yanakakis, see e.g. [PS98]).

A natural question is whether it is possible to obtain a good approximation for the *size* of the optimal vertex cover in time that is *sublinear* in the size of the graph G. Since achieving a pure multiplicative approximation is easily seen to require linear time, we focus on algorithms that compute an estimate $\widehat{\mathrm{VC}}(G)$ such that with high constant probability, $\mathrm{VC}_{\mathrm{opt}}(G) \leq \widehat{\mathrm{VC}}(G) \leq \alpha \cdot \mathrm{VC}_{\mathrm{opt}}(G) + \epsilon n$, for $\alpha \geq 1$ and $0 \leq \epsilon < 1$, where $\mathrm{VC}_{\mathrm{opt}}(G)$ denotes the minimum size of a vertex cover in G. We refer to such an estimate $\widehat{\mathrm{VC}}(G)$ as an (α, ϵ)-*estimate* of $\mathrm{VC}_{\mathrm{opt}}(G)$. Observe that in the special case when the vertex cover is very large, namely $\mathrm{VC}_{\mathrm{opt}}(G) = \Theta(n)$ (which happens for example when the maximum degree and the average degree are of the same order), then an (α, ϵ)-estimate yields an $(\alpha + O(\epsilon))$-multiplicative approximation.

Since an algorithm with complexity sublinear in the size of the graph cannot even read the entire graph, it must have *query access* to the graph. In this work we consider two standard models of queries. In the first model, the algorithm may query the degree $\deg(v)$ of any vertex v of its choice, and it may also query the i^{th} neighbor of v (where the the order on the neighbors is arbitrary). In the second model, more appropriate when the graph is stored as an adjacency matrix, the algorithm can check in a single query whether there is an edge between two vertices v and w chosen by the algorithm. We focus on the first model, but we eventually show that our algorithm can be modified to work in the second model as well.

Previous work. The aforementioned question was first posed by Parnas and Ron [PR07], who showed how to obtain a $(2, \epsilon)$-estimate (for any given additive approximation parameter ϵ) in time $d^{O(\log d/\epsilon^3)}$, where d is the maximum degree in the graph. The dependence on the maximum degree d can actually be replaced by a dependence on $\bar{d}/\epsilon$, where $\bar{d}$ is the average degree in the graph [PR07]. The upper bound of $d^{O(\log d/\epsilon^3)}$ was significantly improved in a sequence of papers [MR09, NO08, YYI09], where the best result due to Yoshida, Yamamoto, and Ito [YYI09] (who analyze an algorithm proposed by Nguyen and Onak [NO08]) is an upper bound of $O(d^4/\epsilon^2)$. Their analysis can also easily be adapted to give an upper bound of $O(\bar{d}^4/\epsilon^4)$ for graphs with bounded average vertex degree $\bar{d}$.

On the negative side, it was also proved in [PR07] that at least a linear dependence on the average degree, $\bar{d}$, is necessary. Namely, $\Omega(\bar{d})$ queries are necessary for obtaining an (α, ϵ)-estimate for any $\alpha \geq 1$ and $\epsilon < 1/4$, provided that $\bar{d} = O(n/\alpha)$, and in particular this is true for $\alpha = 2$. We also mention that obtaining a $(2 - \gamma, \epsilon)$-estimate for any constant γ requires a number of queries that grows at least as the square root of the number of vertices [PR07, due to Trevisan].

Our Result. In this work we describe and analyze an algorithm that computes a $(2, \epsilon)$-estimate of $\mathrm{VC}_{\mathrm{opt}}(G)$ in time $\tilde{O}(\bar{d}) \cdot \mathrm{poly}(1/\epsilon)$. Note that since the graph contains $\bar{d}n/2$ edges, our running time is sublinear for all values of $\bar{d}$. In particular, for graphs of constant average degree, the running time is independent of the number of nodes and edges in the graph, whereas for general graphs it is bounded by at most the square root of the number of edges. In view of the aforementioned lower bound of $\Omega(\bar{d})$, our algorithm is optimal in terms of the dependence on the average degree up to a polylogarithmic factor. Since our algorithm builds on previous work, and in particular on the algorithm proposed and analyzed in [NO08, YYI09],

we describe the latter algorithm first.[1] We refer to this algorithm as **Approx-VC-I**.

The Algorithm Approx-VC-I. Recall that the size of a minimum vertex cover is lower-bounded by the size of any (maximal) matching in the graph, and is upper-bounded by twice the size of any maximal matching. This is indeed the basis of the aforementioned factor-two approximation algorithm, which runs in linear-time. To estimate the size of an arbitrary such maximal matching, the algorithm follows the sampling paradigm of Parnas and Ron [PR07]. That is, the algorithm **Approx-VC-I** selects, uniformly, independently and at random, $\Theta(d^2/\epsilon^2)$ edges. For each edge selected, it calls a *maximal matching oracle*, which we describe momentarily, where the oracle's answers indicate whether or not the edge is in the maximal matching $\mathcal{M}$, for some arbitrary maximal matching $\mathcal{M}$ (that is not a function of the queries to the oracle). The algorithm then outputs an estimate of the size of the maximal matching $\mathcal{M}$ (and hence of a minimum vertex cover) based on the fraction of sampled edges for which the maximal matching oracle returned a positive answer. The number of sampled edges ensures that with high constant probability, the additive error of the estimate is $O((\epsilon/d)m) \leq \epsilon n$, where m is the number of edges in the graph.

The main idea of the algorithm follows the idea suggested in [NO08] which is to simulate the answers of the standard sequential greedy algorithm. The greedy algorithm supposes a fixed ranking (ordering) of the edges in G, which uniquely determines a maximal matching as follows: proceeding along the edges according to the order determined by the ranking, add to the matching each edge that does not share an endpoint with any edge previously placed in the matching. The maximal matching oracle essentially emulates this procedure while selecting a random ranking "on the fly", but is able to achieve great savings in running time by noting that to determine whether an edge is placed in the matching, it is only necessary to know whether or not adjacent edges that are ranked lower than the current edge have been placed in the matching. Namely, given an edge (u, v), it considers all edges that share an endpoint with (u, v) and whose (randomly assigned) ranking is lower than that of (u, v). If there are no such edges, then the oracle returns TRUE. Otherwise it performs recursive calls to these edges, where the order of the calls is according to their ranking. If any recursive call is answered TRUE, then the answer on (u, v) is FALSE, while if all answers (on the incident edges with a lower rank) is answered FALSE, then the answer on (u, v) is TRUE.

Though the correctness of the above algorithm follows directly from the correctness of the greedy algorithm, the query and runtime analysis are more difficult. The analysis of [NO08] is based on a counting argument that shows that it is unlikely that there is a long path of recursive calls with a monotone decreasing set of ranks. Their bound gives a runtime that is independent of the size of the graph, but exponential in the degree d. However, using that the algorithm recurses according to the smallest ranked neighbor, [YYI09] give an ingenious analysis that bounds by $O(d)$ the total number of expected recursive calls when selecting an edge uniformly at random, and when selecting a ranking uniformly at random. This is what allows [YYI09] to obtain an algorithm whose query complexity and running time are $O(d^4/\epsilon^2)$.

Our Algorithm. In what follows we describe an algorithm that has almost linear dependence on the maximum degree d. The transformation to an algorithm whose complexity depends on the average degree $\bar{d}$ is done on a high level along the lines described in [PR07]. We first depart from **Approx-VC-I** by performing the following variation. Rather than sampling edges and approximating the size of a maximal matching by

[1] Yoshida et al. [YYI09] actually analyze an algorithm for approximating the size of a maximal independent set. They then apply it to the line graph of a given graph G, so as to obtain an estimate of the size of a maximal matching, and hence of a minimum vertex cover (with a multiplicative cost of 2 in the quality of the estimate). For the sake of simplicity, we describe their algorithm directly for a maximal matching (minimum vertex cover).

calling the maximal matching oracle on the sampled edges, we sample vertices (as in [PR07]), and we call a *vertex cover oracle* on each selected vertex v. The vertex cover oracle calls the maximal matching oracle on the edges incident to v according to the order induced by their ranking (where the ranking is selected randomly). Once some edge returns TRUE, the vertex cover oracle returns TRUE, and if all incident edges return FALSE, the vertex cover oracle returns FALSE. By performing this variation we can take a sample of vertices that has size $\Theta(1/\epsilon^2)$ rather than[2] $\Theta(d^2/\epsilon^2)$.

Unfortunately, the analysis of [YYI09] is no longer applicable as is. Recall that their analysis bounds the expected number of recursive calls to the maximal matching oracle, for a random ranking, and for a *randomly selected edge*. In contrast, we select a random vertex and call the maximal matching oracle on its (at most d) incident edges. Nonetheless, we are able to adapt the analysis of [YYI09] and give a bound of $O(d)$ on the expected number of recursive calls to the maximal matching oracle, when selecting a vertex uniformly at random.[3]

As a direct corollary of the above we can get an algorithm whose query complexity and running time grow quadratically with d. Namely, whenever the maximal matching oracle is called on a new edge (u, v), the algorithm needs to perform recursive calls on the edges incident to u and v, in an order determined by their ranking. To this end it can query the $O(d)$ neighbors of u and v, assign them (random) rankings, and continue in a manner consistent with the assigned rankings.

To reduce the complexity of the algorithm further, we show a method that for most of the edges that we visit, allows us to query only a small subset of adjacent edges. Ideally, we would like to make only k queries when k recursive calls are made. One of the problems that we encounter here is that if we do not query all adjacent edges, then for some edge (u, v), we could visit a different edge incident to u and a different edge incident to v and make conflicting decisions about the ranking of (u, v) from the point of view of these edges. This could result in an inconsistent execution of the algorithm with results that are hard to predict. Instead, we devise a probabilistic procedure, that, together with appropriate data structures, allows us to perform queries almost "only when needed" (we elaborate on this in the next paragraph). By this we mean that we perform queries only on a number of edges that is a $\mathrm{poly}(\log(d/\epsilon))$ factor larger than the total number of recursive calls made to the maximal matching oracle. We next discuss our general approach.

As in previous work, we implement the random ranking by assigning numbers to edges independently, uniformly at random from $(0, 1]$ (or, more precisely, from an appropriate discretization of $(0, 1]$). For each vertex we keep a copy of a data structure that is responsible for generating and assigning random numbers to incident edges. For each vertex, we can ask the corresponding data structure for the incident edge with the i^{th} lowest number. How does the data structure work? Conceptually, the edges attached to each vertex are grouped into "layers", where the edges in the first layer have random numbers that are at most $1/d$, the edges in layer $i > 1$ have random numbers in the range $2^{i-1}/d$ to $2^i/d$. The algorithm randomly chooses edges to be in a layer for each vertex, one layer at a time, starting with the lowest layer. Each successive layer is processed only as needed by the algorithm. If the algorithm decides that an edge is in the current layer, then it picks a random number for the edge uniformly from the range associated with the layer. In particular, it is possible to ensure that the final random number comes from the uniform distribution on $(0, 1]$. In order to make sure that the same decision is made at both endpoints of an edge (u, v), the data structures for u and v communicate whenever they want to assign a specific random number to the edge. The algorithm

[2] We note that it is actually possible to save one factor of d without changing the algorithm **Approx-VC-I** by slightly refining the probabilistic analysis. This would reduce the complexity of **Approx-VC-I** to cubic in d.

[3] To be more precise, we first give a bound that depends on the ratio between the maximum and minimum degrees as well as on the average degree, and later we show how to obtain a dependence on d (at an extra cost of $1/\epsilon$) by slightly modifying the input graph.

works in such a way so that vertices need query their incident edges only when a communication regarding the specific edge occurs. Our final algorithm is obtained by minimizing the amount of communication between different data structures, and therefore, making them discover not many more edges than necessary for recursive calls in the graph exploration.

Other Related Work. For some restricted classes of graphs it is possible to obtain a $(1, \epsilon)$-estimate of the size of the minimum vertex cover in time that is a function of only ϵ. Elek shows that this is the case for graphs of subexponential growth [Ele10]. For minor-free graphs, one obtains this result by applying the generic reduction of Parnas and Ron [PR07] to local distributed algorithm of Czygrinow, Hańćkowiak, and Wawrzyniak [CHW08]. Both of these results are generalized by Hassidim et al. [HKNO09] to any class of hyperfinite graphs. In particular, for planar graphs, they give an algorithm that computes a $(1, \epsilon)$-estimate in $2^{\mathrm{poly}(1/\epsilon)}$ time. While the running time must be exponential in $1/\epsilon$, unless there exists a randomized subexponential algorithm for SAT, it remains a neat open question whether the query complexity can be reduced to polynomial in $1/\epsilon$.

For bipartite graphs, a $(1, \epsilon n)$-estimate can be computed in $d^{O(1/\epsilon^2)}$ time. This follows from the relation between the maximum matching size and the minimum vertex size captured by König's theorem and fast approximation algorithms for the maximum matching size [NO08, YYI09].

Ideas similar to those discussed in this paper are used to construct sublinear time estimations of other parameters of sparse combinatorial objects, such as maximum matching, set cover, constraint satisfaction [NO08, YYI09, Yos11]. In the related setting of property testing, sublinear time algorithms are given for testing any class of graphs with a fixed excluded minor and any property of graphs with a fixed excluded minor [CSS09, BSS08, Ele10, HKNO09, NS11].

There are also other works on sublinear algorithms for various other graph measures such as the minimum weight spanning tree [CRT05, CS09, CEF$^+$05], the average degree [Fei06, GR08], and the number of stars [GRS10].

2 The Oracle-Based Algorithm

Let $G = (V, E)$ be an undirected graph with n vertices and m edges, where we allow G to contain parallel edges and self-loops. Let d denote the maximum degree in the graph, and let $\bar{d}$ denote the average degree. Consider a *ranking* $\pi : E \to [m]$ of the edges in $G = (V, E)$. As noted in the introduction, such a ranking determines a maximal matching $\mathrm{M}^\pi(G)$. Given $\mathrm{M}^\pi(G)$, we define a vertex cover $\mathrm{C}^\pi(G)$ as the set of all endpoints of edges in $\mathrm{M}^\pi(G)$. Therefore, $\mathrm{VC}_{\mathrm{opt}} \leq |\mathrm{C}^\pi(G)| \leq 2\mathrm{VC}_{\mathrm{opt}}$, where $\mathrm{VC}_{\mathrm{opt}}$ is the minimum size of a vertex cover in G. We assume without loss of generality that there are no isolated vertices in G, since such vertices need not belong to any vertex cover. We shall use the shorthand M^π and C^π for $\mathrm{M}^\pi(G)$ and $\mathrm{C}^\pi(G)$, respectively, when G is clear from the context.

Assume we have an oracle VO^π for a vertex cover based on a ranking π of the edges, where $\mathrm{VO}^\pi(v) =$ TRUE if $v \in \mathrm{C}^\pi(G), \mathrm{VO}^\pi(v) =$ FALSE otherwise. The next lemma follows by applying an additive Chernoff bound.

Lemma 2.1 *For any fixed choice of π, let $\mathrm{C} = \mathrm{C}^\pi(G)$. 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*. With high constant probability,*

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

Algorithm 1, provided below, implements an oracle VO^π, that given a vertex v, decides whether $v \in \mathrm{C}^\pi$. This oracle uses another oracle, MO^π (described in Algorithm 2) that given an edge e, decides whether $e \in \mathrm{M}^\pi$. Both oracles can determine $\pi(e)$ for any edge e of their choice. The oracle MO^π essentially emulates the greedy algorithm for finding a maximal matching (based on the ranking π). We assume that once the oracle for the maximal matching decides whether an edge e belongs to M^π or not, it records this information in a data structure that allows to retrieve it later. By Lemma 2.1, if we perform $\Theta(1/\epsilon^2)$ calls to VO^π, we can get an estimate of the size of the vertex cover C^π up to an additive error of $(\epsilon/2)n$, and hence we can obtain a $(2, \epsilon)$-estimate (as defined in the introduction) of the size of a minimum vertex cover in G. Hence our focus is on upper bounding the query complexity and running time of the resulting approximation algorithm when π is selected uniformly at random.

Algorithm 1: An oracle $\mathbf{VO}^{\boldsymbol{\pi}}(\boldsymbol{v})$ for a vertex cover based on a ranking π of the edges. Given a vertex v, the oracle returns TRUE if $v \in \mathrm{C}^\pi$ and it returns FALSE otherwise.

```
Let e_1, ..., e_t be the edges incident to the vertex v in order of increasing rank (that is,
  π(e_{i+1}) > π(e_i)).
for i = 1, ..., t do
    if MO^π(e_i) = TRUE then
        return TRUE
    return FALSE
```

Algorithm 2: An oracle $\mathbf{MO}^{\boldsymbol{\pi}}(\boldsymbol{e})$ for a maximal matching based on a ranking π of the edges. Given an edge e, the oracle returns TRUE if $e \in \mathrm{M}^\pi$ and it returns FALSE otherwise.

```
if MO^π(e) has already been computed then
    return the computed answer.
Let e_1, ..., e_t be the edges that share an endpoint with e, in order of increasing rank (that is,
  π(e_{i+1}) > π(e_i)).
i ← 1.
while π(e_i) < π(e) do
    if MO^π(e_i) = TRUE then
        return FALSE
    else
        i ← i + 1.
return TRUE
```

We start (in Section 3) by bounding the expected number of calls made to the maximal-matching oracle MO^π in the course of the execution of a call to the vertex-cover oracle VO^π. This bound depends on the average degree in the graph and on the ratio between the maximum degree and the minimum degree. A straightforward implementation of the oracles would give us an upper bound on the complexity of the

algorithm that is a factor of d larger than our final near-optimal algorithm. In Section 4 we describe a sophisticated method of simulating the behavior of the oracle MO^π for randomly selected ranking π, which is selected "on the fly". Using this method we obtain an algorithm with only a polylogarithmic overhead (as a function of d) over the number of recursive calls. Thus, for graphs that are close to being regular, we get an algorithm whose complexity is $\tilde{O}(d/\epsilon^2)$. In Section 5 we address the issue of variable degrees, and in particular, show how to get a nearly-linear dependence on the average degree.

3 Bounding the Expected Number of Calls to the Maximal-Matching Oracle

For a ranking π of the edges of a graph G and a vertex $v \in V$, let $N(\pi, v) = N_G(\pi, v)$ denote the number of different edges e such that a call $\mathrm{MO}^\pi(e)$ was made to the maximal matching oracle in the course of the computation of $\mathrm{VO}^\pi(v)$. Let Π denote the set of all rankings π over the edges of G. Our goal is to bound the expected value of $N(\pi, v)$ (taken over a uniformly selected ranking π and vertex v). We next state our first main theorem.

Theorem 3.1 *Let G be a graph with m edges and average degree $\bar{d}$, and let the ratio between the maximum degree and the minimum degree in G be denoted by ρ. The average value of $N(\pi, v)$ taken over all rankings π and vertices v is $O(\rho \cdot \bar{d})$. That is:*

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

If the graph is (close to) regular, then the bound we get in Theorem 3.1 is $O(\bar{d}) = O(d)$. However, for graphs with varying degrees the bound can be $\Theta(d^2)$. As noted previously, we later show how to deal with variable degree graphs without having to pay a quadratic cost in the maximum degree.

As noted in the introduction, our analysis builds on the work of Yoshida et al. [YYI09]. While our analysis does not reduce to theirs[4], it uses many of their ideas. We start by making a very simple but useful observation about the maximal matching oracle MO^π (Algorithm 2), which follows immediately from the definition of the oracle.

Observation 3.2 *For any edge e, consider the execution of MO^π on e. If in the course of this execution, a recursive call is made to MO^π on another edge e', then necessarily $\pi(e') < \pi(e)$. Therefore, for any consecutive sequence of (recursive) calls to edges $e_\ell, \ldots, e_1$, $\pi(e_\ell) > \pi(e_{\ell-1}) > \ldots > \pi(e_1)$.*

In order to prove Theorem 3.1 we introduce more notation. For any edge $e \in E$, we arbitrarily label its endpoints by $v_a(e)$ and $v_b(e)$ (where if e is a self-loop then $v_a(e) = v_b(e)$, and if e and e' are parallel edges, then $v_a(e) = v_a(e')$ and $v_b(e) = v_b(e')$). For a ranking π and an index k, let π_k denote the edge e such that $\pi(e) = k$.

We say that an edge e is *visited* if a call is made on e either in the course of an oracle computation of $\mathrm{VO}^\pi(v_a(e))$ or $\mathrm{VO}^\pi(v_b(e))$ (that is, as a non-recursive call), or in the course of an oracle computation of

[4]Indeed, we initially tried to find such a reduction. The main difficulty we encountered is that the vertex cover oracle, when executed on a vertex v, performs calls to the maximal matching oracle on the edges incident to v until it gets a positive answer (or all the incident edges return a negative answer). While the analysis of [YYI09] gives us an upper bound on the expected number of recursive calls for a given edge, it is not clear how to use such a bound without incurring an additional multiplicative cost that depends on the degree of the vertices.

$\mathrm{MO}^\pi(e')$ for an edge e' that shares an endpoint with e (as a recursive call). For a vertex v and an edge e, let $X^\pi(v,e) = X^\pi_G(v,e)$ equal 1 if e is visited in the course of the execution of $\mathrm{VO}^\pi(v)$. Using the notation just introduced, we have that

$$N(\pi, v) \;=\; \sum_{e \in E} X^\pi(v,e) \;. \tag{2}$$

Observation 3.3 *Let $e = (v,u)$. If $X^\pi(v,e) = 1$, then for each edge e' that shares the endpoint v with e and for which $\pi(e') < \pi(e)$ we have that $\mathrm{MO}^\pi(e') =$ FALSE.*

To verify Observation 3.3, assume, contrary to the claim, that there exists an edge e' as described in the observation and $\mathrm{MO}^\pi(e') =$ TRUE. We first note that by Observation 3.2, the edge e cannot be visited in the course of an execution of MO^π on any edge $e'' = (v,w)$ such that $\pi(e'') < \pi(e)$ (and in particular this is true for $e'' = e'$). Since $\mathrm{VO}^\pi(v)$ performs calls to the edges incident to v in order of increasing rank, if $\mathrm{MO}^\pi(e') =$ TRUE, then $\mathrm{VO}^\pi(v)$ returns TRUE without making a call to $\mathrm{MO}^\pi(e)$. This contradicts the premise of the observation that $X^\pi(v,e) = 1$.

The next notation is central to our analysis. For $k \in [m]$ and a fixed edge e:

$$X_k(e) \stackrel{\text{def}}{=} \sum_{\pi \in \Pi} \Big(X^\pi(v_a(\pi_k), e) + X^\pi(v_b(\pi_k), e) \Big) \;. \tag{3}$$

That is, $X_k(e)$ is the total number of calls made to the maximal matching oracle on the edge e when summing over all rankings π, and performing an oracle call to the vertex-cover oracle from one of the endpoints of π_k. Observe that

$$\sum_{k=1}^{m} X_k(e) \;=\; \sum_{\pi \in \Pi} \sum_{v \in V} \deg(v) \cdot X^\pi(v,e) \tag{4}$$

where $\deg(v)$ denotes the degree of v in the graph, and for simplicity of the presenation we count each self-loop as contributing 2 to the degree of the vertex. We next give an upper bound on $X_k(e)$.

Lemma 3.4 *For every edge e and every $k \in [m]$:*

$$X_k(e) \leq 2(m-1)! + (k-1) \cdot (m-2)! \cdot d \;. \tag{5}$$

In order to prove Lemma 3.4, we establish the following lemma.

Lemma 3.5 *For every edge e and every $k \in [m-1]$:*

$$X_{k+1}(e) - X_k(e) \leq (m-2)! \cdot d \;. \tag{6}$$

Before proving Lemma 3.5, we show how Lemma 3.4 easily follows from it.

Proof of Lemma 3.4: We prove the lemma by induction on k. For the base case, $k = 1$,

$$X_1(e) = \sum_{\pi} \Big(X^\pi(v_a(\pi_1), e) + X^\pi(v_b(\pi_1), e) \Big) \;. \tag{7}$$

By the definition of the vertex-cover oracle, when starting from either $v_a(\pi_1)$ or from $v_b(\pi_1)$, only a single call is made to the maximal matching oracle. This call is on the edge π_1, which returns TRUE without making

any further calls, because all edges (that share an endpoint with π_1) have a larger rank. This implies that if $e = \pi_1$, then $X^\pi(v_a(\pi_1), e) = X^\pi(v_b(\pi_1), e) = 1$, and otherwise $X^\pi(v_a(\pi_1), e) = X^\pi(v_b(\pi_1), e) = 0$. For any fixed e, the number of rankings π such that $e = \pi_1$ is simply $(m-1)!$ and so $X_1(e) = 2(m-1)!$, as required.

We now turn to the induction step. Assuming the induction hypothesis holds for $k - 1 \geq 1$, we prove it for $k > 1$. This follows directly from Lemma 3.5 (and the induction hypothesis):

$$\begin{aligned} X_k(e) &\leq X_{k-1}(e) + (m-2)! \cdot d && (8)\\ &\leq 2(m-1)! + (k-2) \cdot (m-2)! \cdot d + (m-2)! \cdot d && (9)\\ &= 2(m-1)! + (k-1) \cdot (m-2)! \cdot d\,, && (10) \end{aligned}$$

and the lemma is established. ■

Proof of Lemma 3.5: Throughout the proof we fix k and e. For a ranking π, let π' be defined as follows: $\pi'_{k+1} = \pi_k$, $\pi'_k = \pi_{k+1}$ and $\pi'_j = \pi_j$ for every $j \notin \{k, k+1\}$.

Observation 3.6 *If π and π' are as defined above, then for each edge e where $\pi(e) < k$ (and therefore, $\pi'(e) < k$): $\mathrm{MO}^\pi(e) = \mathrm{MO}^{\pi'}(e)$.*

Observation 3.6 is true due to the fact that if $\pi(e) < k$ then by the definition of π', we have that $\pi'(e) = \pi(e)$. Since in a recursive call we only go to an edge with a lower rank (see Observation 3.2), we get that the execution of $\mathrm{MO}^\pi(e)$ is equivalent to the execution of $\mathrm{MO}^{\pi'}(e)$.

We shall use the notation Π_k for those rankings π in which π_k and π_{k+1} share a common endpoint. Note that if $\pi \in \Pi_k$, then $\pi' \in \Pi_k$ as well (and if $\pi \notin \Pi_k$, then $\pi' \notin \Pi_k$). For two edges $e = (v_1, v_2)$ and $e' = (v_2, v_3)$ (which share a common endpoint v_2), we let $v_c(e, e') = v_c(e', e) = v_2$ ('c' for 'common') and $v_d(e, e') = v_1$, $v_d(e', e) = v_3$ ('d' for 'different'). If e and e' are parallel edges, then we let $v_d(e, e') = v_d(e', e)$ be $v_a(e) = v_a(e')$ and $v_c(e, e') = v_c(e', e)$ be $v_b(e) = v_b(e')$. If e is a self-loop on a vertex v_1 that is also an endpoint of e' (so that $v_2 = v_1$), then $v_d(e, e') = v_c(e, e') = v_1$.

For any edge e and for $1 \leq k \leq m-1$,

$$\begin{aligned} &X_{k+1}(e) - X_k(e)\\ &= \sum_\pi \Big(X^\pi(v_a(\pi_{k+1}), e) + X^\pi(v_b(\pi_{k+1}), e)\Big) - \sum_\pi \Big(X^\pi(v_a(\pi_k), e) + X^\pi(v_b(\pi_k), e)\Big) && (11)\\ &= \sum_{\pi \in \Pi_k} \Big(X^\pi(v_a(\pi_{k+1}), e) + X^\pi(v_b(\pi_{k+1}), e)\Big) - \sum_{\pi \in \Pi_k} \Big(X^\pi(v_a(\pi_k), e) + X^\pi(v_b(\pi_k), e)\Big)\\ &\quad + \sum_{\pi \notin \Pi_k} \Big(X^\pi(v_a(\pi_{k+1}), e) + X^\pi(v_b(\pi_{k+1}), e)\Big) - \sum_{\pi \notin \Pi_k} \Big(X^\pi(v_a(\pi_k), e) + X^\pi(v_b(\pi_k), e)\Big) && (12)\\ &= \sum_{\pi \in \Pi_k} X^\pi(v_c(\pi_{k+1}, \pi_k), e) - \sum_{\pi \in \Pi_k} X^\pi(v_c(\pi_k, \pi_{k+1}), e) && (13)\\ &\quad + \sum_{\pi \in \Pi_k} X^\pi(v_d(\pi_{k+1}, \pi_k), e) - \sum_{\pi \in \Pi_k} X^\pi(v_d(\pi_k, \pi_{k+1}), e) && (14)\\ &\quad + \sum_{\pi \notin \Pi_k} \Big(X^\pi(v_a(\pi_{k+1}), e) + X^\pi(v_b(\pi_{k+1}), e)\Big) - \sum_{\pi \notin \Pi_k} \Big(X^\pi(v_a(\pi_k), e) + X^\pi(v_b(\pi_k), e)\Big). && (15) \end{aligned}$$

By the definition of $v_c(\cdot,\cdot)$, for every $\pi \in \Pi_k$ we have that $v_c(\pi_{k+1},\pi_k) = v_c(\pi_k,\pi_{k+1})$ and so

$$X^{\pi}(v_c(\pi_{k+1},\pi_k),e) = X^{\pi}(v_c(\pi_k,\pi_{k+1}),e)\ , \tag{16}$$

implying that the expression in Equation (13) evaluates to 0. Since $\pi' \in \Pi_k$ if and only if $\pi \in \Pi_k$, we get that

$$\sum_{\pi\in\Pi_k} X^{\pi}(v_d(\pi_{k+1},\pi_k),e) = \sum_{\pi'\in\Pi_k} X^{\pi'}(v_d(\pi'_{k+1},\pi'_k),e) = \sum_{\pi\in\Pi_k} X^{\pi'}(v_d(\pi'_{k+1},\pi'_k),e)\ , \tag{17}$$

and

$$\begin{aligned}\sum_{\pi\notin\Pi_k}\Big(X^{\pi}(v_a(\pi_{k+1}),e) + X^{\pi}(v_b(\pi_{k+1}),e)] &= \sum_{\pi'\notin\Pi_k}\Big(X^{\pi'}(v_a(\pi'_{k+1}),e) + X^{\pi'}(v_b(\pi'_{k+1}),e)\Big)\\ &= \sum_{\pi\notin\Pi_k}\Big(X^{\pi'}(v_a(\pi'_{k+1}),e) + X^{\pi'}(v_b(\pi'_{k+1}),e)\Big)\ .\end{aligned} \tag{18}$$

Therefore,

$$\begin{aligned}X_{k+1}(e) - X_k(e) \;=\; &\sum_{\pi\in\Pi_k} X^{\pi'}(v_d(\pi'_{k+1},\pi'_k),e) - \sum_{\pi\in\Pi_k} X^{\pi}(v_d(\pi_k,\pi_{k+1}),e)\\ &+ \sum_{\pi\notin\Pi_k}\Big(X^{\pi'}(v_a(\pi'_{k+1}),e) + X^{\pi'}(v_b(\pi'_{k+1}),e)\Big)\\ &- \sum_{\pi\notin\Pi_k}\Big(X^{\pi}(v_a(\pi_k),e) + X^{\pi}(v_b(\pi_k),e)\Big)\ .\end{aligned} \tag{19}$$

The next useful observation is that for every $\pi \notin \Pi_k$ (and for every e and $j \in \{a,b\}$),

$$X^{\pi'}(v_j(\pi'_{k+1}),e) = X^{\pi}(v_j(\pi_k),e)\ . \tag{20}$$

This follows by combining the fact that $v_j(\pi'_{k+1}) = v_j(\pi_k)$ with Observations 3.2 and 3.6.

By combining Equation (19) with Equation (20) we obtain that

$$X_{k+1}(e) - X_k(e) \;=\; \sum_{\pi\in\Pi_k} X^{\pi'}(v_d(\pi'_{k+1},\pi'_k),e) - \sum_{\pi\in\Pi_k} X^{\pi}(v_d(\pi_k,\pi_{k+1}),e)\ . \tag{21}$$

Therefore, we need only consider executions in which the underlying rankings π and π' belong to Π_k, and the execution starts from the vertex $v_1(\pi') = v_d(\pi'_{k+1},\pi'_k) = v_d(\pi_k,\pi_{k+1})$. We shall use the shorthand notation $v_2(\pi') = v_c(\pi'_{k+1},\pi'_k) = v_c(\pi_k,\pi_{k+1})$, and $v_3(\pi') = v_d(\pi'_k,\pi'_{k+1}) = v_d(\pi_{k+1},\pi_k)$. For an illustration, see Figure 1. We shall make use of the following simple observation.

Observation 3.7 *Let e be a self-loop. For any vertex v and ranking π, if in the course of the execution of $\mathrm{VO}^{\pi}(v)$ a call is made to $\mathrm{MO}^{\pi}(e)$, then $\mathrm{MO}^{\pi}(e) =$* TRUE.

Observation 3.7 is true since if a call is made to $\mathrm{MO}^{\pi}(e)$ where e is a self-loop, i.e., $e = (v,v)$ for some vertex v, then from Observation 3.3 we know that all other edges incident to v with ranks smaller than $\pi(e)$ return FALSE. Therefore, by the definition of MO^{π} we get that $\mathrm{MO}^{\pi}(e) =$ TRUE.

We would like to understand when $X^{\pi'}(v_1(\pi'),e) = 1$ while $X^{\pi}(v_1(\pi'),e) = 0$. We consider three possible cases (for an illustration see Figure 2) :

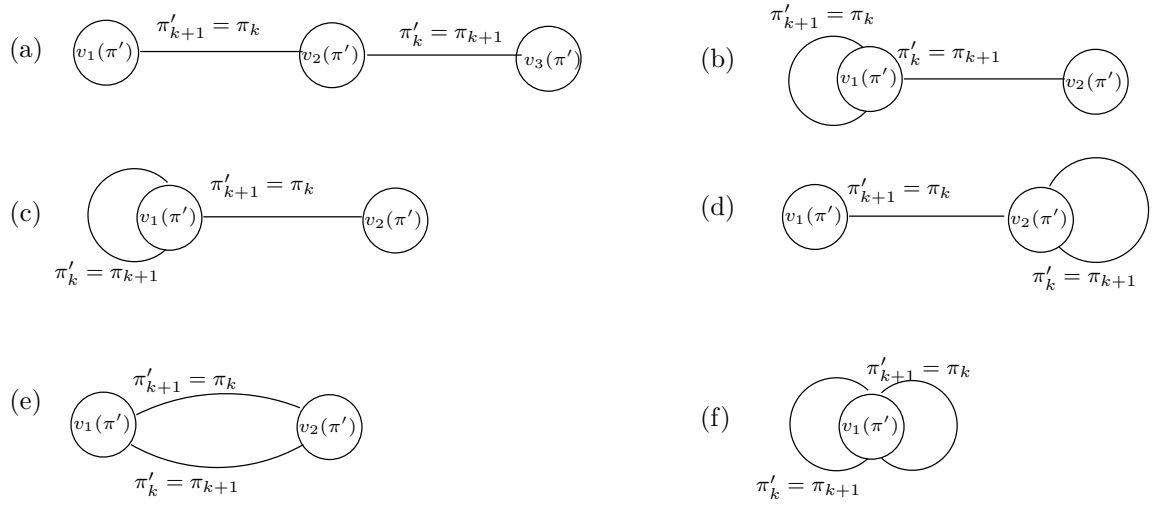

Figure 1: An illustration for the various cases in which $\pi \in \Pi_k$ (i.e., π'_k and π'_{k+1} share at least one endpoint) and we need to compare the executions of $\mathrm{VO}^{\pi}(v_1(\pi'))$ and $\mathrm{VO}^{\pi'}(v_1(\pi'))$ (where $v_1(\pi') = v_d(\pi'_{k+1}, \pi'_k) = v_d(\pi_k, \pi_{k+1})$). We refer to the different cases (a)–(f) in the analysis.

1. $e = (v_1(\pi'), v_2(\pi'))$ (so that $\pi'(e) = k+1$ and $\pi(e) = k$). In this case, if $X^{\pi'}(v_1(\pi'), e) = 1$, then $X^{\pi}(v_1(\pi'), e) = 1$. To verify this, note that if $X^{\pi'}(v_1(\pi'), e) = 1$ then by Observation 3.3, $\mathrm{MO}^{\pi'}(e') = \textsc{false}$ for each edge e' where $v_1(\pi')$ is one of its endpoints and $\pi'(e') < k+1$. By applying Observation 3.6 we get that for each edge e' such that $\pi(e') < k$ we have that $\mathrm{MO}^{\pi}(e') = \mathrm{MO}^{\pi'}(e')$. Therefore, for each edge e' such that $\pi(e') < k$ and $v_1(\pi')$ is one of its endpoints we have that $\mathrm{MO}^{\pi}(e') = \mathrm{MO}^{\pi'}(e') = \textsc{false}$. Hence $X^{\pi}(v_1(\pi'), e) = 1$.

   We note that if π'_k is a self-loop (see cases (c) and (f) in Figure 1), then by Observation 3.7 we have that $\mathrm{MO}^{\pi'}(\pi'_k) = \textsc{true}$. By the definition of $\mathrm{VO}^{\pi'}$ this implies that $\pi'_{k+1} = e$ will not be visited in the course of the execution of $\mathrm{VO}^{\pi'}(v_1(\pi'))$, so that $X^{\pi'}(v_1(\pi'), e)$ is necessarily 0.

2. $e = (v_2(\pi'), v_3(\pi'))$, (so that $\pi(e) = k+1$ and $\pi'(e) = k$). In this case it is possible (though not necessary) that $X^{\pi'}(v_1(\pi'), e) = 1$ and $X^{\pi}(v_1(\pi'), e) = 0$.

3. $e \notin \{(v_1(\pi'), v_2(\pi')), (v_2(\pi'), v_3(\pi'))\}$. In this case it is also possible (though not necessary) that $X^{\pi'}(v_1(\pi'), e) = 1$ and $X^{\pi}(v_1(\pi'), e) = 0$.

   Out of all cases illustrated in Figure 1, this is possible only in cases (a) and (b). We next explain why it is not possible in all other cases.

   - Case (c). If $\mathrm{VO}^{\pi'}(v_1(\pi'))$ visits e before it visits π'_k, then so does $\mathrm{VO}^{\pi}(v_1(\pi'))$ (from Observation 3.6). Otherwise, $\mathrm{VO}^{\pi'}(v_1(\pi'))$ visits π'_k first, but since it is a self-loop, from Observation 3.7 we have that $\mathrm{MO}^{\pi'}(\pi'_k) = \textsc{true}$. By the definition of $\mathrm{VO}^{\pi'}$ we get that $X^{\pi'}(v_1(\pi'), e) = 0$.
   - Case (d). If $\mathrm{VO}^{\pi'}(v_1(\pi'))$ visits e before it visits π'_{k+1}, then so does $\mathrm{VO}^{\pi}(v_1(\pi'))$ (from Observation 3.6). Otherwise, if $\mathrm{VO}^{\pi'}(v_1(\pi'))$ visits π'_{k+1} and e in the same sequence of recursive calls without visiting π'_k, then so does $\mathrm{VO}^{\pi}(v_1(\pi'))$. If there is no such sequence, then $\mathrm{VO}^{\pi'}(v_1(\pi'))$ will visit π'_{k+1} and π'_k. Since π'_k is a self-loop, from Observation 3.7 we have that

$\mathrm{MO}^{\pi'}(\pi'_k) = \textsc{true}$, implying that $\mathrm{MO}^{\pi'}(\pi'_{k+1}) = false$. Therefore, the sequence of recursive calls that visits e in the execution of $\mathrm{VO}^{\pi'}(v_1(\pi'))$, starts from an edge incident to $v_1(\pi')$ whose rank is greater than $k+1$, and the same sequence of calls is made in the execution of $\mathrm{VO}^{\pi}(v_1(\pi'))$.

- Case (e). Since the edges are parallel, if there is a sequence of recursive calls that visits e in the execution of $\mathrm{VO}^{\pi'}(v_1(\pi'))$, then there is such a sequence in the execution of $\mathrm{VO}^{\pi}(v_1(\pi'))$, where the only difference is that the first sequence includes π'_k while the second includes π_k (which are parallel edges).
- Case (f). If $\mathrm{VO}^{\pi'}(v_1(\pi'))$ visits e in a sequence of recursive calls that starts with an edge having rank smaller than k, then from Observation 3.6 so will $\mathrm{VO}^{\pi}(v_1(\pi'))$. Otherwise, since π'_k is a self-loop, by Observation 3.7, if a call is made to $\mathrm{MO}^{\pi'}(\pi'_k)$, then it returns TRUE, causing the execution of $\mathrm{VO}^{\pi'}(v_1(\pi'))$ to terminate without visiting any additional edges (so that e cannot be visited in a sequence of recursive calls that starts with an edge having rank at least k).

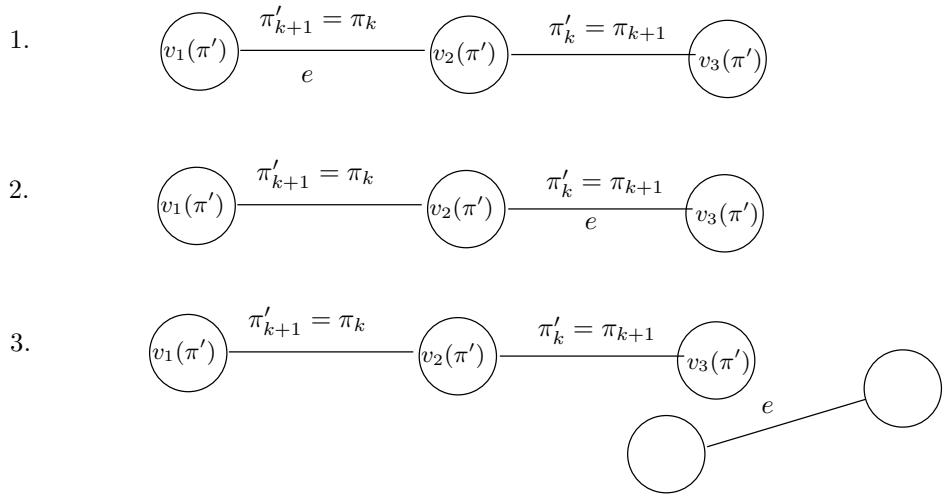

Figure 2: An illustration for the three possible (sub-)cases when $\pi' \in \Pi_k$: 1. $e = (v_1(\pi'), v_2(\pi'))$; 2. $e = (v_2(\pi'), v_3(\pi'))$; 3. $e \notin \{(v_1(\pi'), v_2(\pi')), (v_2(\pi'), v_3(\pi'))\}$. This illustration corresponds to Case (a) in Figure 1 (i.e., no self-loops and no parallel edges).

For a fixed edge e we shall use the following notation for the sets of rankings that correspond to the last two cases described above. Specifically:

- Let $\Pi^{e,1} = \Pi_k^{e,1}$ denote the set of all rankings $\pi' \in \Pi_k$ where $e = (v_2(\pi'), v_3(\pi'))$ and $X^{\pi'}(v_1(\pi'), e) = 1$. (Here we shall make the worst case assumption that $X^{\pi}(v_1(\pi'), e) = 0$).
- Let $\Pi^{\neg e} = \Pi_k^{\neg e}$ denote the set of all rankings $\pi' \in \Pi_k$ where $e \notin \{(v_1(\pi'), v_2(\pi')), (v_2(\pi'), v_3(\pi'))\}$ and $X^{\pi'}(v_1(\pi'), e) = 1$ while $X^{\pi}(v_1(\pi'), e) = 0$.

Thus, $X_{k+1}(e) - X_k(e) \leq |\Pi^{e,1}| + |\Pi^{\neg e}|$. In order to upper bound $|\Pi^{e,1}| + |\Pi^{\neg e}|$, we consider another set of rankings:

- Let $\Pi^{e,0} = \Pi_k^{e,0}$ denote the set of all rankings $\pi' \in \Pi_k$ such that $e = (v_2(\pi'), v_3(\pi'))$ and $X^{\pi'}(v_1(\pi'), e) = 0$.

By the definition of $\Pi^{e,1}$ and $\Pi^{e,0}$, we have that

$$|\Pi^{e,1}| + |\Pi^{e,0}| \leq (m-2)! \cdot d \,. \tag{22}$$

This is true since each ranking $\pi' \in \Pi^{e,1} \cup \Pi^{e,0}$ is determined by first setting $\pi'(e) = k$, then selecting another edge incident to the endpoint $v_2(\pi')$ of e (if e is a self-loop then $v_2(\pi') = v_1(\pi')$) and giving it rank $k+1$ (where there are at most $\deg(v_2(\pi')) - 1 \leq d-1$ such edges), and finally selecting one of the possible $(m-2)!$ rankings for the remaining edges. We next prove that $|\Pi^{\neg e}| \leq |\Pi^{e,0}|$, from which Lemma 3.5 follows.

To this end we prove the next claim.

Claim 3.8 *There is an injection from* $\Pi^{\neg e}$ *to* $\Pi^{e,0}$.

The proof of Claim 3.8 is very similar to a proof of a corresponding claim in [YYI09], but due to our need to extend the proof to a graph with self-loops and parallel edges, and also due to several additional differences, we include it here for completeness.

Proof: We start by making the following observations:

Observation 3.9 *If* $\pi' \in \Pi^{\neg e}$ *and we are in Case (a) as illustrated in Figure 1, then in the course of the execution of* $\mathrm{VO}^{\pi'}(v_1(\pi'))$ *there is a consecutive sequence of recursive calls that includes* π'_{k+1}, π'_k *and* e *at the end. That is, there is a sequence of recursive calls corresponding to a path of edges* $(e_\ell, e_{\ell-1} \ldots e_1)$ *such that* $e_\ell = \pi'_{k+1}, e_{\ell-1} = \pi'_k$ *and* $e_1 = e$.

To verify Observation 3.9, note that since $\pi' \in \Pi^{\neg e}$ we know that $X^{\pi'}(v_1(\pi'), e) = 1$ and $X^{\pi}(v_1(\pi'), e) = 0$. The only difference between the execution of $\mathrm{VO}^{\pi'}(v_1(\pi'))$ and $\mathrm{VO}^{\pi}(v_1(\pi'))$ is that $\mathrm{MO}^{\pi'}(\pi'_{k+1})$ can call $\mathrm{MO}^{\pi'}(\pi'_k)$ but $\mathrm{MO}^{\pi}(\pi'_{k+1}) = \mathrm{MO}^{\pi}(\pi_k)$ cannot call $\mathrm{MO}^{\pi}(\pi'_k) = \mathrm{MO}^{\pi}(\pi_{k+1})$. Thus, the only way that $\mathrm{VO}^{\pi'}(v_1(\pi'))$ and $\mathrm{VO}^{\pi}(v_1(\pi'))$ will create different sequences of recursive calls is when $\mathrm{VO}^{\pi'}(v_1(\pi'))$ calls $\mathrm{MO}^{\pi'}(\pi'_{k+1})$ and then $\mathrm{MO}^{\pi'}(\pi'_{k+1})$ calls $\mathrm{MO}^{\pi'}(\pi'_k)$. Furthermore, these two calls have to be one after the other, since by Observation 3.2, the ranks can only decrease in a sequence of recursive calls.

Observation 3.10 *If* $\pi' \in \Pi^{\neg e}$ *and we are in Case (b) as illustrated in Figure 1, then in the course of the execution of* $\mathrm{VO}^{\pi'}(v_1(\pi'))$ *there is a consecutive sequence of recursive calls that starts with* π'_k*, and ends with* e *(so that, in particular, it does not include* π'_{k+1}*). That is, there is a sequence of recursive calls corresponding to a path of edges* $(e_{\ell-1} \ldots e_1)$ *such that* $e_{\ell-1} = \pi'_k$ *and* $e_1 = e$.

To verify Observation 3.10, note that since $\pi' \in \Pi^{\neg e}$ we know that $X^{\pi'}(v_1(\pi'), e) = 1$ and $X^{\pi}(v_1(\pi'), e) = 0$. The execution of $\mathrm{VO}^{\pi'}(v_1(\pi'))$ cannot visit e in the course of a sequence of recursive calls starting from an edge incident to $v_1(\pi')$ where the edge has ranking smaller k. Otherwise, from Observation 3.6 we would get that $\mathrm{VO}^{\pi}(v_1(\pi'))$ also visits e which contradicts the premise that $\pi' \in \Pi^{\neg e}$. We also know that $\mathrm{VO}^{\pi'}(v_1(\pi'))$ cannot visit π'_{k+1}. If it would have, then since it is a self-loop, from Observation 3.7, $\mathrm{MO}^{\pi'}(\pi'_{k+1}) = \textsc{true}$, causing $\mathrm{VO}^{\pi'}(v_1(\pi'))$ to terminate without visiting e, which contradicts $X^{\pi'}(v_1(\pi'), e) = 1$.

We shall now prove Claim 3.8. Let π^1 be a ranking in $\Pi^{\neg e}$ (so that $\pi^1(e) \notin \{k, k+1\}$). By the definition of $\Pi^{\neg e}$ and by Observations 3.9 and 3.10, we have the following. In Case (a), the execution of $\mathrm{VO}^{\pi^1}(v_1(\pi^1))$ induces a sequence of (recursive) calls to the maximal matching oracle, where this sequence corresponds to a path $P = (e_\ell, \ldots, e_1)$ such that $e_\ell = \pi^1_{k+1}$, $e_{\ell-1} = \pi^1_k$, and $e_1 = e$. In Case (b), the execution of $\mathrm{VO}^{\pi^1}(v_1(\pi^1))$ induces a sequence of (recursive) calls to the maximal matching oracle, where this sequence corresponds to a path $P' = (e_{\ell-1}, \ldots, e_1)$ such that $e_{\ell-1} = \pi^1_k$, and $e_1 = e$. Since in Case (b) P is also a path in the graph, we may refer to the path P in both cases (and take into account, if needed, that in Case (b) $e_\ell = \pi^1_{k-1}$ is a self-loop and is not part of the sequence of recursive calls that reaches e). While we do not know the rankings of the edges $e_{\ell-2}, \ldots e_1$, we know from Observation 3.2 that they are in monotonically decreasing order, and that they are all smaller than k. We also know that the path does not include any parallel edges. This is true since if e_t and e_{t-1} are adjacent edges in the path P and they are parallel edges, then from Observation 3.11 $\pi'(e_{t-1}) < \pi'(e_t)$. But since they are parallel, they have the same endpoints, therefore, by the definition of $\mathrm{VO}^{\pi'}$ and of $\mathrm{MO}^{\pi'}$, the vertex/edge from which the call $\mathrm{MO}^{\pi'}(e_t)$ was made, would have called $\mathrm{MO}^{\pi'}(e_{t-1})$. Furthermore, with the exception of π'_{k+1} in Case (b), the the only edge along the path P that might be a self-loop is e. Otherwise, from Observation 3.7, the self-loop will return true, and thus path P will not visit e.

We can write P as $P = (\pi^1_{\sigma(\ell)}, \ldots \pi^1_{\sigma(1)})$ where $\sigma(i) = \pi^1(e_i)$, so that $\sigma(\ell) = k+1$ and $\sigma(\ell-1) = k$. We next define a mapping φ between rankings, such that $\varphi(\pi^1) = \pi^0$, where we shall show that $\pi^0 \in \Pi^{e,0}$, and that φ is one-to-one. The ranking π^0 is defined as follows by "rotating" the ranks of the edges on P (and leaving the ranks of all other edges as in π^1). Namely, $\pi^0(e_2) = k+1$, $\pi^0(e_1) = k$, and $\pi^0(e_j) = \sigma(j-2)$ for every $3 \leq j \leq \ell$. For an illustration, see Table 1. We first verify that φ is a projection from $\Pi^{\neg e}$ to $\Pi^{e,0}$.

	e_ℓ	$e_{\ell-1}$	$\ldots$	e_3	e_2	$e_1 = e$
Rank in π^1	$\sigma(\ell) = k+1$	$\sigma(\ell-1) = k$	$\ldots$	$\sigma(3)$	$\sigma(2)$	$\sigma(1)$
Rank in π^0	$\sigma(\ell-2)$	$\sigma(\ell-3)$	$\ldots$	$\sigma(1)$	$\sigma(\ell) = k+1$	$\sigma(\ell-1) = k$

Table 1: Ranking of $P = (e_\ell, \ldots, e_1)$ in π^1 and in π^0

Namely, we need to show that:

- $\pi^0 \in \Pi_k$, i.e., π^0_{k+1} and π^0_k share an endpoint $v_2(\pi^0)$, and $e = (v_2(\pi^0), v_3(\pi^0))$.
- $X^{\pi^0}(v_1(\pi^0), e) = 0$ (that is, the execution of $\mathrm{VO}^{\pi^0}(v_1(\pi^0))$ does not create a call to $\mathrm{MO}^{\pi^0}(e)$). In other words, (the execution of) $\mathrm{VO}^{\pi^0}(v_1(\pi^0))$ does not visit e.

The first item directly follows from the definition of π^0. We thus turn to the second item. Recall that by our notational convention, $v_1(\pi^0) = v_d(\pi^0_{k+1}, \pi^0_k) = v_d(e_2, e_1)$ (i.e, it is the endpoint that e_2 does not share with e_1) so that it is the common endpoint of e_2 and e_3, i.e., $v_c(e_2, e_3)$. Since

$$\pi^0(e_3) = \sigma(1) < \sigma(\ell) = k+1 = \pi^0(e_2)\ , \tag{23}$$

the execution of $\mathrm{VO}^{\pi^0}(v_1(\pi^0))$ will visit e_3 before visiting e_2. Since $\pi^0(e) = k$, during the execution of $\mathrm{VO}^{\pi^0}(v_1(\pi^0))$, the call to $\mathrm{MO}^{\pi^0}(e_3)$ will not cause a recursive call to $\mathrm{MO}^{\pi^0}(e)$.

Observe that in the execution of $\mathrm{VO}^{\pi^1}(v_1(\pi^1))$, the call to $\mathrm{MO}^{\pi^1}(e_3)$ creates a recursive call on e_2 (since e_2 follows e_3 on the path P). Therefore, it must be the case that $\mathrm{MO}^{\pi^1}(e')$=FALSE for every e' that has a common endpoint with e_3 and such that $\pi^1(e') < \sigma(2)$. By the definition of φ, all edges that are not

on the path P have the same ranks in π^0 and in π^1. Therefore, all edges with rank lower than $\sigma(1)$ have the same rank in π^1 and in π^0. It follows that for every e' that has a common endpoint with e_3 and such that $\pi^1(e') < \sigma(2)$, $\mathrm{MO}^{\pi^0}(e') = \textsc{false}$. We can conclude that $\mathrm{MO}^{\pi^0}(e_3) = \textsc{true}$ and so $\mathrm{VO}^{\pi^0}(v_1(\pi^0))$ returns TRUE without visiting $e_1 = e$, as required.

It remains to show that φ is an injection from $\Pi^{\neg e}$ to $\Pi^{e,0}$. Assume, contrary to the claim, that φ is not an injection. That is, there are two different rankings $\pi^1 \neq \pi^2 \in \Pi^{\neg e}$ where $\varphi(\pi^1) = \varphi(\pi^2)$. Let $P^1 = (e^1_{\ell_1}, e^1_{\ell_1-1} \ldots e^1_1)$ and $P^2 = (e^2_{\ell_2}, e^2_{\ell_2-1} \ldots e^2_1)$ be the paths that correspond to the sequence of recursive calls to the maximal matching oracle, in the executions of $\mathrm{VO}^{\pi^1}(v_1(\pi^1))$ and $\mathrm{VO}^{\pi^2}(v_1(\pi^2))$ respectively, where $e^1_1 = e^2_1 = e$, $\pi^1(e^1_{\ell_1}) = \pi^2(e^2_{\ell_2}) = k+1$ and $\pi^1(e^1_{\ell_1-1}) = \pi^2(e^2_{\ell_2-1}) = k$ (recall that if π^1 corresponds to Case (b), then $e^1_{\ell_1}$ is a self-loop and is not actually part of the sequence of recursive calls that reaches e, and an analogous statement holds for π^2). Let s be the largest index such that $(e^1_s, e^1_{s-1} \ldots e^1_1) = (e^2_s, e^2_{s-1} \ldots e^2_1)$. We denote this common subsequence by $(e_s, e_{s-1} \ldots e_1)$. Observe that $s \geq 2$. This is true since: (1) By the definitions of the paths, $e^1_1 = e^2_1 = e$, and (2) given that $\varphi(\pi^1) = \varphi(\pi^2) = \pi^0$ and $\pi^0(e^1_2) = \pi^0(e^2_2) = k+1$, it holds that $e^1_2 = e^2_2$.

By the definitions of φ and s we have that $\pi^1(e_i) = \pi^2(e_i)$ for each $i \in [s-2]$. Thus, $\sigma_1(i) = \sigma_2(i)$ for each $i \in [s-2]$, where we shall sometimes use the shorthand $\sigma(i)$ for this common value. For an illustration, see Table 2.

	Rank from $\varphi(\pi^1)$	Rank from $\varphi(\pi^2)$
$\pi^0(e_1)$	$\sigma_1(\ell_1 - 1) = k$	$\sigma_2(\ell_2 - 1) = k$
$\pi^0(e_2)$	$\sigma_1(\ell_1) = k+1$	$\sigma_2(\ell_2) = k+1$
$\pi^0(e_3)$	$\sigma_1(1)$	$\sigma_2(1)$
$\vdots$	$\vdots$	$\vdots$
$\pi^0(e_{s-1})$	$\sigma_1(s-3)$	$\sigma_2(s-3)$
$\pi^0(e_s)$	$\sigma_1(s-2)$	$\sigma_2(s-2)$

Table 2: Ranks of edges $e^1_1 = e^2_1 \ldots e^1_{s-2} = e^2_{s-2}$ are equal in π^1 and π^2

.

The next observation will be useful.

Observation 3.11 *For every edge e', if $\pi^1(e') < \min\{\sigma_1(s-1), \sigma_2(s-1)\}$ or $\pi^2(e') < \min\{\sigma_1(s-1), \sigma_2(s-1)\}$, then $\pi^1(e') = \pi^2(e')$. Therefore, $\mathrm{MO}^{\pi^1}(e') = \mathrm{MO}^{\pi^2}(e')$ for e' such that $\pi^1(e') = \pi^2(e') < \min\{\sigma_1(s-1), \sigma_2(s-1)\}$.*

We consider two cases:

1. P^2 is a suffix of P^1 or P^1 is a suffix of P^2. Without loss of generality, assume that P^2 is a suffix of P^1, so that $s = \ell_2$.

2. Otherwise (neither path is a suffix of the other), assume without loss of generality that $\sigma_1(s-1) < \sigma_2(s-1)$.

In both cases, since e^1_{s+1} is not on P^2, φ, when applied to π^2 does not change the ranking of e^1_{s+1}. That is, $\pi^0(e^1_{s+1}) = \pi^2(e^1_{s+1})$. Since (by the definition of φ) $\pi^0(e^1_{s+1}) = \sigma_1(s-1)$, we get that

$$\pi^2(e^1_{s+1}) = \sigma_1(s-1) = \pi^1(e^1_{s-1}) \ . \tag{24}$$

In the first case (where P^2 is a suffix of P^1), we have that $\sigma_2(s-1) = k$, while $\sigma_1(s-1) < k$, and so

$$\sigma_1(s-1) < \sigma_2(s-1) \ (= \pi^2(e^2_{s-1})) \ . \tag{25}$$

In the second case, this inequality was made as an explicit assumption.

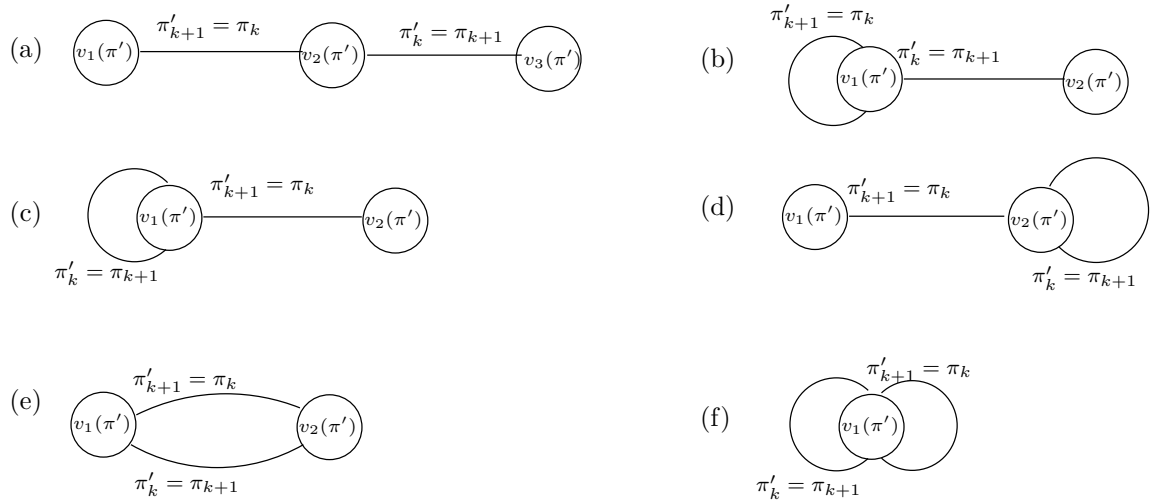

Figure 3: An illustration for the proof of Claim 3.8.

We thus have that the execution of $\mathrm{MO}^{\pi^2}(e^2_s)$ visits e^1_{s+1} before visiting e^2_{s-1}. We would like to understand what occurs in the call to $\mathrm{MO}^{\pi^2}(e^1_{s+1})$. If we are in Case (b) and $P^1 = (\pi^1_{k+1}, \pi^1_k, e)$, i.e., $s = k$, then, since $e^1_{s+1} = \pi^1_{k+1}$ is a self-loop, from Observation 3.7, $\mathrm{MO}^{\pi^2}(e^1_{s+1}) = \textsc{true}$. Hence $\mathrm{MO}^{\pi^2}(e^2_s = e^1_s)$ returns FALSE without visiting $e^2_{s-1} = e$, but this stands in contradiction to the definition of P^2. If we are in Case (a), then since the path P^1 corresponds to a sequence of recursive calls to the maximal-matching oracle, we have that for every edge e' that shares an end-point with e^1_{s+1} and such that $\pi^1(e') < \sigma_1(s) = \pi^1(e^1_s)$, the call to $\mathrm{MO}^{\pi^1}(e')$ returns FALSE. Combining this with Observation 3.11, we get that for every edge e' that shares an end-point with e^1_{s+1} and such that $\pi^2(e') < \sigma_1(s)$, the call to $\mathrm{MO}^{\pi^2}(e')$ returns FALSE. By Equation (24) we get that $\mathrm{MO}^{\pi^2}(e^1_{s+1})$ returns TRUE. Hence, $\mathrm{MO}^{\pi^2}(e^2_s = e^1_s)$ returns FALSE without visiting e^2_{s-1}, but this stands in contradiction to the definition of P^2. ■ (Claim 3.8)
Having established Claim 3.8, the proof of Lemma 3.5 is completed. ■

We are now ready to prove Theorem 3.1.

Proof of Theorem 3.1: Recall that d denotes the maximum degree, $\bar{d}$ denotes the average degree and ρ denotes the ratio between the maximum degree and the minimum degree, which is denoted by $d_{\min}$ (where the latter is at least 1 since we assumed without loss of generality that there are no isolated vertices). By combining Equations (2) and (4) and applying Lemma 3.4 (as well as recalling that we counted each self-

loop as contributing 2 to the degree of a vertex), we get that:

$$\begin{aligned}
&\frac{1}{m!} \cdot \frac{1}{n} \cdot \sum_{\pi \in \Pi} \sum_{v \in V} N(\pi, v) \\
&\leq \frac{1}{m!} \cdot \frac{1}{n} \cdot \frac{1}{2d_{\min}} \sum_{e \in E} \sum_{k=1}^{m} X_k(e) && (26) \\
&\leq \frac{1}{m!} \cdot \frac{1}{n} \cdot \frac{1}{2d_{\min}} \cdot m \cdot \left(m \cdot 2(m-1)! + \frac{m \cdot m - 1}{2} \cdot (m-2)! \cdot d \right) && (27) \\
&= O\left(\frac{m}{n} \cdot \frac{d}{d_{\min}} \right) = O(\rho \cdot \bar{d}) \,, && (28)
\end{aligned}$$

and we obtain the bound claimed. ■

4 Limiting the Exploration of Neighbor Sets

The analysis in the previous section suffices to show an algorithm whose query complexity and running time are a factor of d larger than the number of oracle calls that it makes. The factor of d in this expression is due to querying all edges that are incident to the endpoints of each edge for which a call to the maximal matching oracle is made (where, as we explain momentarily, a random ranking can be selected in an online fashion).

This section is devoted to a method for selecting incident edges of low rank efficiently without querying entire neighborhoods of relevant vertices. By applying the method, we reduce the query complexity and the running time by a factor of almost d. The main challenges here are to ensure that the ranks of encountered edges indeed come from the uniform distribution over all permutations and that the same decision with respect to a rank of an edge is made at both endpoints of the edges.

Replacing a random ranking with random numbers. The oracle construction described as Algorithm 1 and Algorithm 2 uses a random ranking $\pi : E \to [m]$ of edges. We start by replacing a random ranking of edges with random real numbers in $(0, 1]$ selected uniformly and independently for every edge $e \in E$, yielding a vector $\sigma : E \to (0, 1]$ which we use in the same way as the ranking π. Since the probability that two edges are assigned the same real number is 0, whenever the oracle compares the ranks of two edges e and e', it can check whether $\sigma(e) < \sigma(e')$, instead of whether $\pi(e) < \pi(e')$, effectively yielding a random ranking of edges. Since each $\sigma(e)$ is independent, this small conceptual shift allows one to generate $\sigma(e)$ at random in an easier manner and to simplify the analysis. Though it is not possible to generate and store real numbers in $(0, 1]$, we later introduce a proper discretization.

4.1 A Data Structure for Accessing Neighbors

The oracle described as Algorithms 1 and 2 always collects all edges around the vertex or edge being considered and sorts them to explore them recursively in increasing order of their random numbers. In this section we introduce a data structure that is responsible for generating the random numbers and providing edges for further exploration in the desired order.

For every vertex $v \in V$, we have a copy `neighbors`$[v]$ of the data structure. (In fact, a copy for a given vertex is created when it is accessed for the very first time.) From the point of view of the exploration

algorithm, the data structure *exposes only one operation*: `lowest`(k), where k is a positive integer. The operation `neighbors`$[v]$.`lowest`(k) lists edges incident to v in order of the random numbers assigned to them, omitting all appearances of parallel edges or self-loops except the first one, which has been assigned the lowest number. For each positive k, the operation returns a pair $\langle w, r\rangle$, where w is a vertex and r is a number in $(0, 1] \cup \{\infty\}$. If $r \neq \infty$, then (v, w) is the edge with the k^{th} lowest number in the above order, and r is the number assigned to it. Otherwise, the list is shorter than k and $r = \infty$ indicates the query concerned a non-existing edge. We present the implementation of the data structure in Section 4.4.

We rewrite Algorithms 1 and 2 to use the data structure, and present them as the oracle $\mathrm{VO}^\sigma(v)$ in Algorithm 3 and the oracle $\mathrm{MO}^\sigma(e)$ in Algorithm 4, respectively.

Algorithm 3: An oracle $\mathbf{VO}^{\boldsymbol{\sigma}}(\boldsymbol{v})$ for a vertex cover based on the input from the data structures `neighbors`, which assigns edges $\boldsymbol{e}$ random numbers $\boldsymbol{\sigma}(\boldsymbol{e})$ (online). Given a vertex $\boldsymbol{v}$, the oracle returns TRUE if $\boldsymbol{v}$ belongs to the corresponding vertex cover and it returns FALSE otherwise.

```
1  i:=1
2  ⟨w, r⟩ := neighbors[v].lowest(i)
3  while r ≠ ∞ do
4      if MO^σ((v, w)) = TRUE then
5          return TRUE
6      i := i + 1
7      ⟨w, r⟩ := neighbors[v].lowest(i)
8  return FALSE
```

Algorithm 4: An oracle $\mathbf{MO}^{\boldsymbol{\sigma}}(\boldsymbol{(u, v)})$ for a maximal matching based on the input from the data structures `neighbors`, which assigns edges $\boldsymbol{e}$ random numbers $\boldsymbol{\sigma}(\boldsymbol{e})$ (online). Given an edge $\boldsymbol{(u, v)}$, the oracle returns TRUE if $\boldsymbol{(u, v)}$ belongs to the corresponding matching and it returns FALSE, otherwise.

```
1  if MO^σ((u, v)) has already been computed then
2      return the computed answer
3  k1 := 1 and k2 := 1
4  ⟨w1, r1⟩ := neighbors[u].lowest(k1)
5  ⟨w2, r2⟩ := neighbors[v].lowest(k2)
6  while w1 ≠ v or w2 ≠ u do
7      if r1 < r2 then
8          if MO^σ((u, w1)) = TRUE then return FALSE
9          k1 := k1 + 1
10         ⟨w1, r1⟩ :=neighbors[u].lowest(k1)
11     else
12         if MO^σ((v, w2)) = TRUE then return FALSE
13         k2 := k2 + 1
14         ⟨w2, r2⟩ := neighbors[v].lowest(k2)
15 return TRUE
```

Claim 4.1 *Let σ be an injection from E to $(0,1]$. Let $\pi : E \to [|E|]$ be the corresponding ranking defined in such a way that for every edge e, $\sigma(e)$ is the $\pi(e)^{\text{th}}$ lowest number in the set $\{\sigma(e') : e' \in E\}$.*

For every vertex v, the answer returned by $\mathrm{VO}^{\sigma}(v)$ (Algorithm 3) is the same as the answer returned by $\mathrm{VO}^{\pi}(v)$ (Algorithm 1) provided the operation `lowest` *works as specified and gives answers consistent with σ.*

Proof: It is easy to verify that the claim holds when there are no parallel edges. This is true because when there are no parallel edges, a sequence of calls to $\texttt{neighbors}[v].\texttt{lowest}(1), \ldots, \texttt{neighbors}[v].\texttt{lowest}(k)$ simply returns the first k edges incident to v in order of increasing rank. Furthermore, when called on an edge (u, v), Algorithm 4 effectively merges the two corresponding lists of adjacent edges (i.e., those incident to u and those incident to v) to obtain a single list sorted according to rank, and makes recursive calls in the order dictated by the list.

It remains to verify that the same is true when there are parallel edges. For a fixed choice of σ and the induced ranking π consider the two trees of recursive calls when calling $\mathrm{VO}^{\pi}(v)$ (Algorithm 1) and $\mathrm{VO}^{\sigma}(v)$ (Algorithm 3), where the former calls the oracle M^{π} (Algorithm 2), and the latter calls the oracle M^{σ} (Algorithm 4). When we refer to an edge in in the tree we actually mean an occurence of an edge in G on a path of recursive calls.

These trees are both rooted at v, and with each edge there is an associated rank (number) and an associated answer computed by the corresponding maximal matching oracle. Recall that each path of recursive calls from the root to a leaf passes through edges with decreasing ranks (numbers). Furthermore, in both trees, if an edge (u, v) in the tree is associated with the answer FALSE, then there must be an edge (u, w) (or (v, w)) adjacent to it in the tree with lower rank (a "child" of this edge) that is associated with the answer TRUE, and it is the highest ranking child that (u, v) has. If (u, v) is associated with the answer TRUE, then all the children of (u, v) in the tree are associated with the answer FALSE. It will actually be convenient to consider the full recursion trees without the "memoization" rule that we employ (which says that once an answer is determined for an edge it is not computed again). This in particular implies that for each edge that is the last edge on a path of recursive calls, the answer associated with it must be TRUE.

By the definition of $\mathrm{VO}^{\sigma}(v)$ and the operation `lowest`, the tree corresponding to $\mathrm{VO}^{\sigma}(v)$ contains only edges that have minimal ranking among each set of parallel edges that connect a pair of vertices. We claim that this tree is a "pruned" version of the tree that corresponds to $\mathrm{VO}^{\pi}(v)$, in the sense that all subtrees containing non-minimally ranked parallel edges are removed, and the answers associated with the remaining edges (and hence with the root v) are exactly the same.

Let $T^{\pi}(v)$ denote the tree of recursive calls for $\mathrm{VO}^{\pi}(v)$, and let h be the height of $T^{\pi}(v)$. Starting from $\ell = h$ and going up the tree, we show that we can remove all non-minimally ranked parallel edges in level ℓ of $T^{\pi}(v)$ without altering the answer for their parent edges. For $\ell = h$, we claim that there are no non-minimally ranked parallel edges in the last level of $T^{\pi}(v)$, so that no pruning needs to be performed. To verify this, assume in contradiction that e is a non-minimally ranked parallel edge between vertices u and w where e is at the end of a recursive path of length h in $T^{\pi}(v)$. Since e is not minimally ranked, there should be a "sibling" of e in the tree which correspond to the minimally ranked edge e' between u and w. Since $\pi(e') < \pi(e)$, it must be the case that the answer associated with e', that is, $\mathrm{M}^{\pi}(e')$, is FALSE. But e' also belongs to level h, so that e' is the last edge on a path of recursive calls, and hence cannot be answered FALSE.

Assuming we have performed the pruning successfully for all levels $\ell < \ell' \leq h$, we show that we can perform it for level ℓ. Consider a non-minimally ranked parallel edge e between vertices u and v in level ℓ of $T^{\pi}(v)$. As argued above, there is a "sibling" of e in the tree which correspond to the minimally ranked

edge e' between u and w. Since $\pi(e') < \pi(e)$, it must be the case that the answer associated with e', that is, $\mathrm{M}^{\pi}(e')$, is FALSE. This implies that e' has a child e'' in the tree resulting from pruning all non-minimal parallel edges from levels $\ell' > \ell$, such that $\mathrm{M}^{\pi}(e'') =$ TRUE. But since $\pi(e'') < \pi(e') < \pi(e)$, and e'' is also adjacent to e, we get that $\mathrm{M}^{\pi}(e)$ is FALSE as well. Hence, it is possible to prune e from the tree without altering the answer obtained for its parent. ■

4.2 Implementing `lowest`: The High-Level Idea

The pseudo-code for the procedure `lowest` as well as the data structure that it uses, are given in full detail in Subsection 4.4. Here we give a high-level description. For the sake of simplicity of the presentation, in this description we assume that there are no parallel edges.

Roughly speaking, the procedure `lowest` for a vertex v is implemented in "batches". Namely, considering intervals of $(0, 1]$ of the form $(2^{-i}, 2^{-i+1}]$ (for $i \in [d_\star]$, where $d_\star = \lceil \log d \rceil$, as well as the interval $(0, 2^{-d_\star}]$), the procedure does the following. It first decides which edges incident to v should be assigned a value in the current interval $(2^{-i}, 2^{-i+1}]$. In this stage each edge is identified with its label (in $\{1, \ldots, \deg(v)\}$). The procedure then determines the identity of the other endpoint of each of these edges by performing a neighbor query, and it assigns the edge a value $\sigma((v, w))$, selected uniformly at random from the interval. This assignment is performed unless a certain constraint is discovered due to information held in `neighbors`$[w]$, as we explain subsequently. Once $\sigma((v, w))$ is determined, the other endpoint of the edge, w, is "notified". That is, the data structure `neighbors`$[w]$ is updated with this new information. The procedure "opens" a new interval $(2^{-i+1}, 2^{-i+2}]$ if the index k it is called with is such that the number of neighbors w of v whose identity has been revealed and such that $\sigma((v, w)) \leq 2^{-i+1}$ is strictly less than k. Thus, the procedure performs queries and assigns valued to edges "on demand", but it does so for "batches" of edges. More precise details follow.

The data structure `neighbors` maintains two values for each vertex v: `lb`, and `next_lb` (where the latter is always twice the former). When a vertex is first encountered, `lb` is set to 0 and `next_lb` is set to $2^{-d_\star}$. Second, the data structure maintains a dictionary `assigned_number`, which holds, for those vertices w that are known to be neighbors of v, the value $\sigma((v, w))$ that was assigned to the edge between them (initially, the dictionary is empty). The subset of indices in $\{1, \ldots, \deg(v)\}$ that correspond to edges for which the other endpoint has not been revealed (and do not yet have an associated value), are considered *unassigned*. Third, the data structure maintains a list of pairs $\langle w, r \rangle$, where w is a (known) neighbor of v and $r = \sigma((v, w))$. This list is sorted in ascending order of r's, and it contains exactly those w for which the corresponding r is at most `lb`.

If a call is made to `neighbors`$[v]$.`lowest`(k) with $k > \deg(v)$ then it returns[5] $\langle v, \infty \rangle$. Otherwise, the procedure does the following until the length of `sorted` is at least k. It first considers those edges (v, w) incident to v that were already assigned a value r and this value belongs to the interval $(\texttt{lb}, \texttt{next_lb}]$ (that is, `assigned_number`$[w] \in (\texttt{lb}, \texttt{next_lb}]$). The setting of the value r for each such edge (v, w) was performed previously in the course of call to `neighbors`$[w]$.`lowest`$(\cdot)$. Let the corresponding subset of pairs $\langle w, r \rangle$ be denoted S.

The procedure next selects a subset T of $\{1, \ldots, \deg(v)\}$ containing the labels of those (additional) edges that it will (tentatively) assign a value in (`lb`,`next_lb`]. Putting aside for now the issue of time-efficiency (which we return to later), this can be done by flipping a coin with bias $\frac{\texttt{next_lb} - \texttt{lb}}{1 - \texttt{lb}}$ indepen-

[5]Recall that we assume that there are no parallel edges, or else $\langle v, \infty \rangle$ is returned if k exceeds the "effective" degree of v, that is, counting parallel edges as a single edge.

dently for each edge label in the subset of unassigned edge labels. For each $t \in T$, the procedure now performs a neighbor query to obtain the t^{th} neighbor of v. Denoting this neighbor by w, let `lb'` denote the lower bound `lb` held by w, that is, in the data structure `neighbors`$[w]$. If `lb`$' \leq$ `lb`, so that the lower bound constraint imposed by w is no larger than that imposed by v, then the following operations are performed.

First, a random number r in the interval (`lb`,`next_lb`] is selected uniformly at random, and `assigned_number`$[w]$ is set to r. In addition, `assigned_number`$[v]$ is set to r in the data structure `neighbors`$[w]$ (so that w is "notified" of the revealed edge (v, w) as well as the assignment $r = \sigma((v, w))$). Finally, the pair $\langle w, r\rangle$ is added to S.

If `lb`$' >$ `lb`, which means that `lb`$' \geq$ `next_lb` (given the way the intervals are defined), then the lower bound constraint imposed by the end point w of the edge (v, w) does not allow the edge to be assigned a value in the interval (`lb`,`next_lb`], and so effectively its selection to T is retracted. Note that since the decision whether an edge label is added to T is done independently for the different edges, the end effect (of not assigning (v, w) a value in (`lb`,`next_lb`]) is exactly the same as the one we would get if we had the knowledge in advance (before selecting T), that the corresponding edge label t should not be selected.

After going over all labels t in T, the resulting set S of pairs $\langle w, r\rangle$ is sorted in ascending order of r's, and it is appended to the end of the list `sorted`. Thus, `sorted` now includes all pairs $\langle w, r\rangle$ such that w is a neighbor of v, the value assigned to this edge is r, and $r \leq$ `next_lb`. The variables `lb` and `next_lb` are then updated so that `lb` is set to `next_lb` and `next_lb` is set to $2 \cdot$ `next_lb`. Once the length of `sorted` is at least k, the procedure returns `sorted`$[k]$. In Subsection 4.4 we formally establish that the distribution of random numbers the data structures `neighbors`$[v]$ provide access to via the operation `lowest`(k) is the same as assigning independently at random a number from the range $(0, 1]$ to each edge.

4.3 Generating Random Numbers

In this subsection we describe a random process that generates random numbers $\sigma(e)$ for edges $e \in E$. The procedure `lowest` applies this process in the course of its executions. In the remainder of this section, $|\mathcal{I}|$ denotes the length of an arbitrary real interval $\mathcal{I}$. We do not distinguish open and closed intervals here. For instance, $|(0, 1)| = |[0, 1]| = |(0, 1]| = |[0, 1)| = 1$.

Let d be an upper bound on the maximum vertex degree. We set $d_\star = \lceil \log d \rceil$. For every edge e, the number $\sigma(e)$ should be selected independently, uniformly at random from the range $(0, 1]$. We partition this range into $d_\star + 1$ intervals. We set

$$\mathcal{I}_i = \begin{cases} (2^{-i}, 2^{-i+1}] & \text{for } i \in [d_\star], \\ (0, 2^{-d_\star}] & \text{for } i = d_\star + 1. \end{cases}$$

Algorithm 5: A Process for Selecting a Random Number Assigned to an Edge

1 **for** $i \leftarrow d_* + 1$ **downto** 2 **do**
2 with probability $\frac{|\mathcal{I}_i|}{\sum_{1 \leq j \leq i} |\mathcal{I}_j|}$: **return** a number selected from $\mathcal{I}_i$ uniformly at random (and terminate)
3 **return** a number selected from $\mathcal{I}_1$ uniformly at random

We describe our process as Algorithm 5. The process first selects one of the intervals $\mathcal{I}_i$, and then selects a number uniformly at random from this interval. The selection of the interval is conducted as follows. We

consider the intervals in reverse order, from $\mathcal{I}_{d_\star+1}$ to $\mathcal{I}_1$. For a considered interval, we decide that the number belongs to this interval with probability equal to the length of the interval over the sum of lengths of all the remaining intervals. The process selects each interval with probability proportional to its length, and since the lengths of all intervals sum up to 1, the number that the process returns is uniformly distributed on the entire interval $(0, 1]$.

Corollary 4.2 *Algorithm 5 selects a random number from the uniform distribution on* $(0, 1]$.

Note that by simulating a few iterations of the loop in the above process, one can decide that the number assigned to a given edge is either a specific number less than or equal to 2^{-i}, or that it is greater than 2^{-i} without specifying it further, for some i. Later, whenever more information about the number is needed, we may continue with consecutive iterations of the loop. As we see later, we use the process in our data structures `neighbors`$[v]$ to lower the query and time complexity of the resulting vertex cover algorithm.

4.4 Data Structures

We now describe the data structures `neighbors`$[v]$. Each data structure `neighbors`$[v]$ simulates the random process described in Section 4.3 for all edges incident to v in the course of the executions of `neighbors`$[v]$.`lowest`. The data structure simultaneously makes a single iteration of the loop in Algorithm 5 for all incident edges. It may be the case that for some edge (v, w), the random number has already been specified. In this case, the result of the iteration for this (v, w) is discarded. It may also be the case that this iteration of the loop has already been taken care of by `neighbors`$[w]$, the data structure for the other endpoint of the edge. The data structures communicate to make sure that a second execution of a given iteration does not overrule the first. The data structures are designed to minimize the amount of necessary communication. Note that if a data structure does not have to communicate with a data structure at the other endpoint of a given neighbor, it does not even have to know the neighbor it is connected to with a given edge, which can be used to save a single query. By using this approach, we eventually save a factor of nearly d in the query complexity.

Each data structure `neighbors`$[v]$ supports the following operations:

`neighbors`$[v]$.`lowest`(k)**:** As already mentioned, this is the only operation that is directly used by the oracles (Algorithm 3 and Algorithm 4). It returns a pair $\langle w, r\rangle$, where (v, w) is the edge with the k^{th} lowest random number assigned to edges incident to v, omitting a second and furher appearances for parallel edges, and r is the random value assigned to (v, w). If $r = \infty$, then k is greater than the length of such a defined list.

`neighbors`$[v]$.`lower_bound`$()$**:** The operation returns the current lower bound the data structure imposes on the edges that are incident to v and have not been assigned a specific random number yet. The set of possible values returned by the procedure is $\{0\} \cup \{2^i : -d_\star \leq i \leq 0\}$. Let ℓ_v be the number returned by the operation. It implies that the data structure simultaneously simulated the random process described in Section 4.3 for incident edges until it made sure that the random numbers that have not been fixed belong to $(\ell_v, 1]$.

Furthermore, let (v, w) be an edge in the graph. Let ℓ_v and ℓ_w be the numbers returned by the operation for `neighbors`$[v]$ and `neighbors`$[w]$, respectively. If no specific random number has been assigned to (v, w), then we know that the random number will eventually be selected uniformly at random from $(\max\{\ell_v, \ell_w\}, 1]$.

`neighbors`$[v]$`.set_value`(w, r)**:** It is used to notify the data structure `neighbors`$[v]$ that the random value assigned to (v, w) has been set to r. This operation is used when the data structure `neighbors`$[w]$ assigns a specific random number to (v, w). Before assigning r, the data structure `neighbors`$[w]$ has to make sure that $r >$ `neighbors`$[v]$`.lower_bound`$()$, i.e., it has not been decided by the data structure `neighbors`$[v]$ that the random number assigned to v is greater than r.

To implement the above operations, each data structure `neighbors`$[v]$ maintains the following information:

`lb`**:** The variable specifies the lower bound on the incident edges that were not assigned a random number yet. This is the value returned by the operation `neighbors`$[v]$`.lower_bound`$()$. This is also the value at which the simulation of the process generating random number for edges incident to v has stopped.

`next_lb`**:** If specific random numbers assigned to more edges are necessary, the next considered range of random numbers will be $($`lb`$,$ `next_lb`$]$, and `next_lb` will become the new lower bound for the edges that have not been assigned any random number. This variable is redundant, because its value is implied by the value of `lb`, but using it simplifies the pseudocode.

`assigned_number`**:** This is a dictionary that maps neighbors w of v to numbers in $(0, 1]$. Initially, the dictionary is empty. If `assigned_number`$[w] =$ NONE, i.e., there is no mapping for w, then no specific random number has been assigned to any of the edges (v, w) yet. Otherwise, `assigned_number`$[w]$ is the lowest random number that has been assigned to any parallel edge (v, w).

`sorted`**:** This is a list consisting of pairs $\langle w, r\rangle$, where w is a neighbor of v and r is the number assigned to the edge (v, w). It is sorted in ascending order of r's, and it contains exactly those w for which the edge (v, w) (with the lowest assigned random number) has an assigned random number less than or equal to `lb`. For all neighbors w that do not appear on the list, the lowest number assigned to any edge (v, w) is greater than `lb`.

We give pseudocode for all data structure operations as Algorithms 6, 7, 8, and 9. We postpone all issues related to an efficient implementation of the data structure to Section 4.6. Three of them are straightforward, and we only elaborate on the operation `neighbors`$[v]$`.lowest`(k) (see Algorithm 9).

Algorithm 6: The procedure for initializing `neighbors`$[v]$

```
1 lb := 0
2 next_lb := 2^{-d_⋆}
3 assigned_number := {empty map}
4 sorted := {empty list}
```

Algorithm 7: The procedure `neighbors`$[v]$`.set_value`(w, r)

```
1 assigned_number[w] := r
```

Algorithm 8: The procedure neighbors[v].lower_bound()

1 **return** lb

As long as not sufficiently many lowest random numbers assigned to edges incident to v have been determined, the operation lowest simulates the next iteration of the loop in the random process that we use for generating random numbers. Let I be the interval $(\texttt{lb}, \texttt{next_lb}]$. The operation wants to determine all random numbers assigned to edges incident to v that lay in I. First, in Line 2, it determines the random numbers in I that have already been assigned by the other endpoints of corresponding edges. In Line 3, the operation simulates an iteration of the loop of the random process for all edges incident to v to determine a subset of them that will have numbers in I (unless it has already been decided for a given edge that its random number is not in I). In the loop in Line 4, the operation considers each of these edges. Let (v, w) be one of them, where w is its other endpoint, queried by the operation. In Line 6, the operation generates a prospective random number $r \in I$ for the edge. First, the operation makes sure that this iteration of the has not been simulated by the other endpoint (the condition in Step 7). If this is the case, the operation considers two further cases. If r is lower than the lowest number assigned to any parallel edge (v, w) so far, the procedure updates the appropriate data structures with this information (Steps 8–11). If no random number has ever been assigned to any edge (v, w), the procedure assigns it and updates the data structures appropriately (Step 12–15). When the operation finishes going over the list of all potentially selected edges and eventually determines all incident edges with new lowest random numbers, it sorts them in order of their random number and appends them in this order to the list sorted. Finally, when sufficiently many edges with lowest numbers have been determined, the operation returns the identity of the edge with the k^{th} smallest number.

Lemma 4.3 *The lists of incident edges that the data structures* neighbors[v] *provide access to via the operation* lowest(k) *are distributed in the same way as when each edge is assigned independently at random a number from the range* $(0, 1]$.

Proof: We know from Corollary 4.2 that the random process generates a random number from the distribution $(0, 1]$. Each data structure neighbors[v] simulates consecutive iterations of the loop in this process for all edges incident to v. Consider a group of parallel edges (v, w). For each of these edges, the random process is simulated by both neighbors[v] and neighbors[w]. We have to show that until the lowest number assigned to the edges in this group is determined (which happens when it is added to the list sorted), then for each edge the decision made in the first simulation matters. Why is this the case? Recall that the random process considers intervals $\mathcal{I}_{d_\star+1}, \mathcal{I}_{d_\star}, \ldots, \mathcal{I}_1$ as the sources of the random number in this order. As long as both neighbors[v] and neighbors[w] reject a given interval their decisions are the same, so the first decision is in effect. Now suppose without loss of generality that neighbors[w] simulates a consecutive iteration of the loop in the random process and decides to use $\mathcal{I}_i$ as the source of the random number for a given edge (v, w) in Step 3 of the operation lowest. If neighbors[v] has already simulated this iteration (the condition verified in Step 7), the operation does not proceed. Otherwise, the random number assigned to the edge is considered for a new minimum random number assigned to this group of parallel edges. Note that since the operation keeps simulating iterations even after a random number is assigned, it could be the case for a specific copy of (v, w) that a new, higher random number is considered, but it is ignored, because it is higher than the first decision, which is the only one that has impact on the list that the operation lowest provides access to.

Algorithm 9: The procedure `neighbors`$[v]$`.lowest`(k)

while length(`sorted`) $< k$ and `lb` < 1 **do**
  S := set of pairs $\langle w, r\rangle$ such that `assigned_number`$[w] = r$ and $r \in$ (`lb`, `next_lb`]
  T := subset of $\{1, \ldots, \deg(v)\}$ with each number included independently
    with probability $\frac{\texttt{next_lb}-\texttt{lb}}{1-\texttt{lb}}$
  foreach $t \in T$ **do**
    $w := t^{\text{th}}$ neighbor of v
    r := a number selected uniformly at random from (`lb`, `next_lb`]
    if `neighbors`$[w]$`.lower_bound()` $\leq$ `lb` **then**
      if $\exists \langle w, r'\rangle \in S$ *s.t.* $r < r'$ **then**
        `assigned_number`$[w] := r$
        `neighbors`$[w]$`.set_value`(v, r)
        replace $\langle w, r'\rangle$ with $\langle w, r\rangle$ in S
      if `assigned_number`$[w] =$ NONE **then**
        `assigned_number`$[w] := r$
        `neighbors`$[w]$`.set_value`(v, r)
        $S := S \cup \{\langle w, r\rangle\}$
  Sort S in ascending order of their r, and append at the end of `sorted`
  `lb` := `next_lb`
  `next_lb` := $2 \cdot$ `next_lb`
if length(`sorted`) $< k$ **then return** $\langle v, \infty\rangle$
else return `sorted`$[k]$

The correctness of the data structure follows from the fact that it extends the list `sorted` by always adding all edges with random numbers in a consecutive interval, and it always takes into consideration decisions already made by data structures for the other endpoints for these intervals. ■

4.5 Query Complexity

We now show that the number of queries that the algorithm makes is not much higher than the number of recursive calls in the graph exploration procedures. The following simple lemma easily follows from the Chernoff bound and will help us analyze the behavior of the algorithm.

Lemma 4.4 *Let $X_1, \ldots, X_s$ be independent random Bernoulli variables such that each X_i equals 1 with probability p. It holds:*

- *For any $\delta \in (0, 1/2)$,*
$$\sum_i X_i \leq 6 \cdot \ln(1/\delta) \cdot \max\{1, ps\}.$$
with probability at least $1 - \delta$.

- *For any $\delta \in (0, 1/2)$, if $ps > 8\ln(1/\delta)$, then*

$$\sum_i X_i \geq \frac{ps}{2}.$$

with probability at least $1 - \delta$.

Proof: Let us first prove the first claim. If $6 \cdot \ln(1/\delta) \cdot \max\{1, ps\} \geq s$, the claim follows trivially. Otherwise, there exist independent Bernoulli random variables Y_i, $1 \leq i \leq s$ such that for each i,

$$\Pr[Y_i = 1] = 3 \cdot \ln(1/\delta) \cdot \max\{1/s, p\} > p$$

since from the definition of δ: $3 \cdot \ln(1/\delta) > 1$. Therefore $\Pr[X_i = 1] < \Pr[Y_i = 1]$. By this fact and by the Chernoff bound,

$$\begin{aligned} \Pr[\textstyle\sum X_i > 2E[\sum Y_i]] &\leq \Pr[\textstyle\sum Y_i > 2E[\sum Y_i]] \\ &\leq \exp(-\ln(1/\delta) \cdot \max\{1, ps\}) \\ &\leq \exp(-\ln(1/\delta)) \leq \delta. \end{aligned}$$

The second claim also directly follows from the Chernoff bound:

$$\Pr[\textstyle\sum X_i < ps/2] \leq \exp(-(1/2)^2 \cdot ps/2) \leq \delta.$$

■

Definition 4.5 *Denote $\mathcal{J}_i = \bigcup_{j=i}^{d_\star+1} \mathcal{I}_j$, where $1 \leq i \leq d_\star + 1$. For example: $\mathcal{J}_1 = (0, 1]$ and $\mathcal{J}_{d_\star+1} = (0, \frac{1}{d}]$. We expect that the number of incident edges to v with random numbers in $\mathcal{J}_i$ to be $\deg(v) \cdot |\mathcal{J}_i|$.*

We now define a property of vertices that is useful in our analysis. Intuitively, we say that a vertex is "usual" if the numbers of incident edges with random numbers in specific subranges of $(0, 1]$ are close to their expectations.

Definition 4.6 *Let $\alpha > 0$. We say that a vertex v is* α-usual *if the random numbers assigned to edges incident to v have the following properties for all $i \in \{1, \ldots, d_\star + 1\}$:*

- *Upper bound: The number of incident edges with random numbers in $\mathcal{J}_i$ is*

$$\textit{at most } \max\{\alpha, \alpha \cdot \deg(v) \cdot |\mathcal{J}_i|\}.$$

- *Lower bound: If $\deg(v) \cdot |\mathcal{J}_i| \geq \alpha$, then the number of edges with random numbers in $\mathcal{J}_i$ is*

$$\textit{at least } \deg(v) \cdot |\mathcal{J}_i|/2.$$

We now basically want to show that the relevant vertices are α-*usual*, and later on we will use it to prove a lower bound.

We define an additional quantity that is useful later in bounding the total running time of the algorithm.

Definition 4.7 *For an execution of Step 3 of Algorithm 9 where the number of neighbors is k and $p \in [0,1]$ is the probability of selecting each of them, we say that the* toll *for running it is kp.*

We now prove a bound on the query complexity of the algorithm and other quantities, which are useful later to bound the running time. We start by introducing the main Lemma (Lemma 4.8), followed by proving Lemma 4.9 which will help us prove Lemma 4.8.

Lemma 4.8 *Consider an algorithm $\mathcal{A}$ that queries the input graph only via the oracle described as Algorithm 1. Let $t \geq 1$ be the expected resulting number of calls in $\mathcal{A}$ to the oracles described as Algorithm 1 and Algorithm 2. Let d be an upper bound on the maximum degree of the input graph.*

Suppose now that we run this algorithm replacing calls to Algorithm 1 with calls to Algorithm 3. The following events hold all at the same time with probability $1 - 1/20$:

1. *The total number of calls to Algorithms 3 and 4 is $O(t)$*
2. *The operation* `lowest` *in data structures* `neighbors`$[v]$ *runs at most $O(t)$ times.*
3. *The query complexity of $\mathcal{A}$ is $O(t \cdot \log^2(dt))$.*
4. *The total toll for running Step 3 of Algorithm 9 is $O(t \cdot \log(dt))$.*

Before proving Lemma 4.8 we establish the following Lemma:

Lemma 4.9 *Assume the conditions of Lemma 4.8. Let $t' = 100t$, $\delta = 1/(40000t(d+1)(d_\star+1))$, and $\alpha = 8 \cdot \ln(1/\delta)$. The following three events happen with probability less than $\frac{1}{100}$ for each:*

1. *The total number of calls to Algorithm 3 and Algorithm 4 is bounded by t'.*
2. *The first $2t'$ vertices for which the operation* `lowest` *is called are α-usual.*
3. *For the first $2t'$ vertices v for which the operation* `lowest` *is called, the size of the set T generated in the j^{th} execution of Step 3 of the operation is bounded by $\alpha \cdot \max\{1, \deg(v) \cdot 2^{j-d_\star}\}$.*

Proof: For every group of parallel edges, the operation `lowest` lists only the edge with the lowest number. For the purpose of this analysis we assume that the operation lists in fact all occurences of a given parallel edge. The final complexity is only reduced because of the fact that some unnecessary calls are omitted.

1. Let us bound the probability that one of the above events does not occur. By Markov's inequality the probability that the first event does not occur is bounded by $\frac{1}{100}$.
2. We shall now prove that the first $2t'$ vertices for which the operation `lowest` is called are α-usual. The total number of vertices that have an incident edge for which the process generating random numbers is simulated in the above calls is bounded by $2t' \cdot (d+1)$. The property of being α-usual is a function of only random numbers assigned to incident edges.
   For $\mathcal{J}_i$ let $X = \sum_{j=1}^{s} X_j$ where $p = \Pr[X_j = 1] = |\mathcal{J}_i|$, $s = \deg(v)$, i.e. X is the number of all incident edges to v with random numbers in $\mathcal{J}_i$. From Lemma 4.4 we get that:

$$\begin{aligned}\Pr\big[\sum_i X_i > \alpha \cdot \max\{1, |\mathcal{J}_i|\deg(v)\}\big] &= \Pr\big[\sum_i X_i > 8 \cdot \ln(1/\delta) \cdot \max\{1, ps\}\big] \\ &\leq \Pr[\sum_i X_i > 6 \cdot \ln(1/\delta) \cdot \max\{1, ps\}] < \delta\end{aligned}$$

Also, from Lemma 4.4 we get that:

$$\Pr\big[\sum_i X_i < \frac{\deg(v)\cdot|\mathcal{J}_i|}{2}\big] \;=\; \Pr\big[\sum_i X_i < \frac{ps}{2}\big] < \delta$$

i.e. v is not α-*usual* because of $\mathcal{J}_i$ with probability less than 2δ. From union bound on all all $i \in [d_\star + 1]$ we get that vertex v is not α-*usual* with probability less than $2\delta(d_\star + 1)$.
Using the union bound again, this time over the vertices incident to edges for which the random process is run, the probability that any of them is not α-usual is bounded by

$$2t' \cdot (d+1) \cdot 2\delta(d_\star + 1) = 400t\delta(d+1)(d_\star+1) = \frac{1}{100}.$$

3. We need to prove that for the first $2t'$ vertices v for which the operation `lowest` is called, the size of the set T generated in the j^{th} execution of Step 3 of the operation is bounded by $\alpha \cdot \max\{1, \deg(v) \cdot 2^{j-d_\star}\}$.
Let v be one of the first $2t'$ vertices for which the operation `neighbors`$[v]$.`lowest` is called. Observe that in the j^{th} iteration of the loop **while**, $(\texttt{next_lb} - \texttt{lb})/(1 - \texttt{lb})$ is at most $2^{j-d_\star}$. Therefore, it follows from Lemma 4.4 that for each $j \in \{1, \ldots, d_\star + 1\}$, the size of the set T in Algorithm 9 selected in the j^{th} execution of Step 3 is bounded by $\alpha \cdot \max\{1, \deg(v) \cdot 2^{j-d_\star}\}$ with probability $1 - \delta$. By the union bound over all j and the first $2t'$ vertices, the probability that the third event does not occur is bounded by

$$2t'(d_\star + 1)\delta = 200t(d_\star + 1) \cdot 1/(40000t(d+1)(d_\star+1) < \frac{1}{100}$$

■

Summarizing, the probability that at least one of the three events does not occur is bounded by

$$\frac{3}{100} < \frac{1}{20}$$

Let us now prove Lemma 4.8 assuming that the events in Lemma 4.9 occur.

Proof of Lemma 4.8:

1. We need to prove that the total number of calls to Algorithms 3 and 4 is $O(t)$. This follows directly from Lemma 4.9, we proved it there for $t' = O(t)$.

2. We need to show that the operation `lowest` in data structures `neighbors`$[v]$ runs at most $O(t)$ times.
The total number of vertices v for which the operation `neighbors`$[v]$.`lowest` is called is bounded by $2t'$, because a call to one of the oracles (Algorithms 3 and 4) requires calling the operation `lowest` for at most two vertices. It follows from the implementation of the oracles that the operation `neighbors`$[v]$.`lowest` is executed at most $3t' = O(t)$ times if the number of oracle calls is bounded by t' (which was proved in Lemma 4.9). This is true because in Algorithm 3 we call `neighbors`$[v]$.`lowest` once and in Algorithm 4 we call `neighbors`$[v]$.`lowest` twice.

3. We will now show that the query complexity of $\mathcal{A}$ is $O(t \cdot \log^2(dt))$.
For each vertex v, denote $k_v \in [0, \deg(v)]$ the number of times we call `neighbors`$[v]$.`lowest`(k) on v. We assume that if the operation is not executed for a given vertex, then $k_v = 0$. It holds that:

$$\sum_{v \in V} k_v \leq 3t'$$

We now attempt to bound the query complexity necessary to execute the operation `neighbors`$[v]$.`lowest` for a given v such that $k_v > 0$. Note that the expected number of edges with random numbers in a given $\mathcal{J}_i$ is $\deg(v)/2^{i-1}$. Recall that from Lemma 4.9 we know that the first $2t'$ vertices for which the operation `lowest` is called are α-usual. From the lower bound of α-usual (Definition 4.5) we get that If $\deg(v) \cdot |\mathcal{J}_i| \geq \alpha$, then the number of edges with random numbers in $\mathcal{J}_i$ is at least

$$\deg(v) \cdot |\mathcal{J}_i|/2.$$

Therefore, if

$$\deg(v) \cdot |\mathcal{J}_i| = \deg(v)/2^{i-1} \geq \max\{2\alpha, 2k_v\}$$

then the number of edges with random numbers in $\mathcal{J}_i$ is at least

$$\frac{\max\{2\alpha, 2k_v\}}{2} = \max\{\alpha, k_v\} \geq k_v$$

i.e. if i is such that $\deg(v)/2^{i-1}$ is at least $\max\{2\alpha, 2k_v\}$, then at least k_v edges incident to v have random numbers in $\mathcal{J}_i$. This also holds for i such that $\deg(v)/2^{i-1} \geq 2\alpha k_v$. Let i_v be the largest integer i such that $2^i \leq \frac{\deg(v)}{\alpha k_v}$ (remember $i = d_{\star+1}, d_\star \cdots$). Since i_v is the maximum i that satisfies this, then

$$2^{i_v+1} > \frac{\deg(v)}{\alpha k_v} \Rightarrow 2^{i_v} > \frac{\deg(v)}{2\alpha k_v} \Rightarrow 2^{-i_v} < \frac{2\alpha k_v}{\deg(v)}$$

The body of the loop **while** in Algorithm 9 is executed at most $d_\star + 2 - i_v$ times for v (remember we start from $i = d_{\star+1}$), independently of how many times the operation is executed for v, because all relevant edges incident to v are discovered during these iterations. From Lemma 4.9 we know that the size of the set T in Algorithm 9 selected in the j^{th} execution of this loop is bounded by $\alpha \cdot \max\{1, \deg(v) \cdot 2^{j-d_\star}\}$. Furthermore, the sum of sizes of all sets T generated for v is bounded by

$$\begin{aligned}\sum_{j=1}^{d_\star+2-i_v} \alpha \cdot \max\{1, \deg(v) \cdot 2^{j-d_\star}\} &\leq \alpha(d_\star + 1) + 2\alpha \cdot \deg(v) \cdot 2^{2-i_v} \\ &\leq \alpha(d_\star + 1) + 16\alpha^2 k_v.\end{aligned}$$

This also bounds the number of neighbor queries for v. Since these are the only neighbor queries in the algorithm, by summing over all v with $k_v \geq 0$, the total number of neighbor queries is bounded by

$$2t' \cdot \alpha(d_\star + 1) + \sum_{v \in V} 16\alpha^2 k_v \leq 200\alpha t(d_\star + 1) + 16\alpha^2 \cdot 300t = O(\alpha t(d_\star + \alpha)) = O(t \cdot \log^2(dt)).$$

(Recall $t' = 200t$ and that $\sum_{v \in V} k_v \leq 3t'$). Note that degree queries appear only in Step 3 of the operation `neighbors`$[v]$.`lowest` with one query to discover the size of the set from which a subset is selected. The number of degree queries is in this case bounded by the total number of executions of Step 3, which is at most $O(t \cdot \log d)$. Summarizing, the total query complexity is $O(t \cdot \log^2(dt))$.

4. Finally, we need to prove that the total toll for running Step 3 of Algorithm 9 is $O(t \cdot \log(dt))$. Recall that the toll is defined as kp where k is the number of neighbors and p is the probability to selecting each of them in an execution of Step 3 of Algorithm 9. Using arguments as above, the toll for running Step 3 in the operation `neighbors`$[v]$`.lowest` for a given v is bounded by

$$\sum_{j=1}^{d_\star+2-i_v} \deg(v) \cdot 2^{j-d_\star} \leq 2 \cdot \deg(v) \cdot 2^{2-i_v} \leq 8 \cdot \deg(v) \cdot \frac{2\alpha k_v}{\deg(v)} = 16\alpha k_v$$

By summing over all vertices v, we obtain a bound on the total toll:

$$\sum_{v \in V} 16\alpha k_v \leq 4800\alpha t = O(t \cdot \log(dt)).$$

■

4.6 Efficient Implementation

We have already introduced techniques that can be used to show an approximation algorithm whose query complexity has near-linear dependence on the maximum degree d. Unfortunately, a straightforward implementation of the algorithm results in a running time with approximately quadratic dependence on d. The goal of this section is to remove a factor of approximately d from the running time of the algorithm. Our main problem is how to efficiently simulate Step 3 in the operation `lowest`. Note that Step 3 is sampling from a binomial distribution.

First, in Lemma 4.11, we prove that there is an algorithm that can simulate a binomial distribution which runs in efficient time. Finally, in Theorem 4.13, we will show how to use it in our algorithms and how to bound the running time by $O(t \cdot \log^3(dt))$.

We start by defining the binomial distribution.

Definition 4.10 *We write $B(k, p)$, where k is a positive integer and $p \in [0, 1]$, to denote the* binomial distribution with success probability p *on $\{0, 1, \ldots, k\}$ distributed as $\sum_{i=1}^{k} X_i$, where each X_i, $1 \leq i \leq k$, is an independent random variable that equals 1 with probability p, and 0 with probability $1 - p$.*

It is well known that the probability that a value drawn from the binomial distribution $B(k, p)$ equals q is $\binom{k}{q} p^q (1-p)^{k-q}$. We now show how to efficiently sample from this distribution.

Lemma 4.11 *Let a, b, k, and Q be positive integers, where $a \leq b$ and $Q > 1$, that can be represented in the standard binary form, using a constant number of machine words. There is an algorithm that takes a, b, k, and Q as parameters, runs in $O(\max\{ka/b, 1\} \cdot \log Q)$ time, and outputs an integer selected from a distribution $\mathcal{D}$ on $\{0, 1, \ldots, k\}$ such that the total variation distance between $\mathcal{D}$ and $B(k, a/b)$ is bounded by $1/Q$.*

Proof: If $a = b$, then the algorithm can return the trivial answer in $O(1)$ time, so we can safely assume for the rest of the proof that $a < b$. Let $p = a/b$ and let $q_i = \binom{k}{i} p^i (1-p)^{k-i}$ be the probability of drawing i from $B(k, p)$. Let $s = \min\{6 \cdot \ln(2Q) \cdot \max\{1, ka/b\}, k\}$. For each $i \leq s$, we compute a real number $q'_i \in [0, 1]$ such that $q_i - q'_i \leq 1/2(k+1)Q$ and $\sum_{i=0}^{s} q'_i = 1$ (details about how to compute those q'_i are

given in Lemma 4.12). Then we select the output of the algorithm from the distribution given by q'_i's. We write $\mathcal{D}$ to denote this distribution.

Let us bound the total variation distance between this distribution and $B(k,p)$. It suffices to show that for every subsets S of $\{0,\ldots,k\}$, the probability of selecting an integer from S in $B(k,p)$ is not greater by more than $1/Q$, compared to the probability of selecting an integer in S from $\mathcal{D}$. Consider an arbitrary such set S. Let S_1 be the subset of S consisting of numbers at most s. Let S_2 be the subset of S consisting of integers greater than s. We have

$$\sum_{i\in S} q'_i \geq \sum_{i\in S_1} q'_i \geq \sum_{i\in S_1}\left(q_i - \frac{1}{2(k+1)Q}\right) \geq \left(\sum_{i\in S_1} q_i\right) - \frac{1}{2Q}. \tag{29}$$

Recall that $X_i = 1$ with probability p. If $s = k$ then

$$\Pr\big[\sum_{i=1}^{k} X_i > s\big] = \big[\sum_{i=1}^{k} X_i > k\big] = 0$$

If $s = 6\cdot\ln(2Q)\cdot\max\{1, ka/b\}$ then we define $\delta = \frac{1}{2Q}$, and from Lemma 4.4 we have that

$$\Pr\big[\sum_{i=1}^{k} > 6\cdot\ln(\frac{1}{\delta})\cdot\max\{1,pk\}\big] < \delta$$

Hence,

$$\Pr\big[\sum_{i=1}^{k} > s\big] < \frac{1}{2Q}$$

In other words: the probability that a number greater than s is being selected from $B(k,p)$ (i.e. s X_i's are 1) is bounded by $\frac{1}{2Q}$. Therefore,

$$\left(\sum_{i\in S_2} q_i\right) < \frac{1}{2Q} \tag{30}$$

From 29 and 30 we get:

$$\sum_{i\in S} q'_i \geq \left(\sum_{i\in S_1} q_i\right) - \frac{1}{2Q} + \left(\sum_{i\in S_2} q_i\right) - \frac{1}{2Q} \geq \left(\sum_{i\in S} q_i\right) - \frac{1}{Q},$$

Therefore,

$$\sum_{i\in S}[q_i - q'_i] \leq \frac{1}{Q},$$

which proves our Lemma. Next, in Lemma 4.12 we will also show that the running time of the algorithm is $O(s) = O(\max\{k\frac{a}{b}, 1\}\log(Q))$. ■

We now describe how to compute values q'_i that are approximation to q_i.

Lemma 4.12 *Recall: $a < b$, $p = a/b$ and $q_i = \binom{k}{i}p^i(1-p)^{k-i}$ (probability of drawing i from $B(k,p)$). Let $s = \min\{6\cdot\ln(2Q)\cdot\max\{1,ka/b\}, k\}$. For each $i \leq s$, we can compute a real number $q'_i \in [0,1]$ such that $q_i - q'_i \leq 1/2(k+1)Q$ and $\sum_{i=0}^{s} q'_i = 1$. The total running time is $O(\max\{ka/b, 1\}\cdot\log Q)$.*

Proof: Observe that for $1 \leq i \leq k$:

$$q_i = q_{i-1} \cdot \frac{k+1-i}{i} \cdot \frac{p}{1-p}$$

Let $t_i = \frac{q_i}{q_0}$ for $0 \leq i \leq s$. It holds that for $1 \leq i \leq s$:

$$t_i = t_{i-1} \cdot \frac{k+1-i}{i} \cdot \frac{p}{1-p} = t_{i-1} \cdot \frac{k+1-i}{i} \cdot \frac{a}{b-a} \tag{31}$$

Note that for $0 \leq i \leq s$:

$$\frac{t_i}{\sum_{j \leq s} t_j} = \frac{\frac{q_i}{q_0}}{\frac{1}{q_0}\sum_{j \leq s} q_j} \geq q_i \tag{32}$$

Suppose now that instead of t_i, we use $t'_i \geq 0$, $0 \leq i \leq s$, such that $|t_i - t'_i| \leq \frac{\max_{0 \leq j \leq s} t_j}{4(k+1)^2 Q}$. Then from the definition of t'_i we get:

$$t'_i \geq t_i - \frac{\max_{0 \leq j \leq s} t_j}{4(k+1)^2 Q} \tag{33}$$

Also:

$$\sum_{j \leq s} t'_j \leq s \cdot \frac{(\max_{0 \leq j \leq s} t_j)}{4(k+1)^2 Q + \sum_{j \leq s} t_j} \leq \frac{\max_{0 \leq j \leq s} t_j}{4(k+1)Q + \sum_{j \leq s} t_j} \leq \big(1 + \frac{1}{4(k+1)Q}\big) \cdot \sum_{j \leq s} t_j \tag{34}$$

We have

$$\begin{aligned}
\frac{t'_i}{\sum_{j \leq s} t'_j} &\geq \frac{t_i - \frac{\max_{0 \leq j \leq s} t_j}{4(k+1)^2 Q}}{(1 + \frac{1}{4(k+1)Q}) \sum_{j \leq s} t_j} \\
\text{(From 32 and that } \max_{0 \leq j \leq s} t_j \leq \textstyle\sum_{j \leq s} t_j) &\geq \frac{q_i \cdot \sum_{j \leq s} t_j}{(1 + \frac{1}{4(k+1)Q}) \cdot \sum_{j \leq s} t_j} - \frac{\frac{\sum_{j \leq s} t_j}{4(k+1)^2 Q}}{(1 + \frac{1}{4(k+1)Q}) \cdot \sum_{j \leq s} t_j} \\
\text{(Since } 1 + \tfrac{1}{4(k+1)^2 Q} \geq 1) &\geq \frac{q_i}{(1 + \frac{1}{4(k+1)Q})} - \frac{1}{4(k+1)^2 Q} \\
&\geq q_i\big(1 - \frac{1}{4(k+1)Q}\big) - \frac{1}{4(k+1)^2 Q} \\
&\geq q_i - \frac{1}{4(k+1)Q} - \frac{1}{4(k+1)^2 Q} \geq q_i - \frac{1}{2(k+1)Q}.
\end{aligned}$$

So eventually we get that

$$q_i - \frac{t'_i}{\sum_{j \leq s} t'_j} \leq \frac{1}{2(k+1)Q} \tag{35}$$

Also, note that $\sum_{i=0}^{s} \frac{t'_i}{\sum_{j=0}^{s} t'_j} = 1$.

Therefore, in our distribution $\mathcal{D}$, we will define $q'_i = \frac{t'_i}{\sum_{j \leq s} t'_j}$.

It remains to show how we obtain t'_i with the desired properties. For this purpose, we use floating-point arithmetic. Each positive number that we obtain during the computation is stored as a pair $\langle S, E \rangle$ representing $S \cdot 2^E$. We require that $2^\alpha \leq S < 2^{\alpha+1}$ and $|E| \leq \beta$, for some α and β to be set later. If we can

perform all standard operations on these integers in $O(1)$ time, then we can perform the operations on the represented positive real numbers in $O(1)$ time as well. We call S a *significand* and E an *exponent*.

In particular, to multiply two numbers $\langle S_1, E_1\rangle$ and $\langle S_2, E_2\rangle$ it suffices to multiply S_1 and S_2, truncate the least significant bits of the product, and set the new exponent accordingly. If these two numbers are multiplicative $(1 \pm \delta_1)$- and $(1 \pm \delta_2)$-approximations to some quantities X_1 and X_2, respectively, then the product of S_1 and S_2 in our arithmetic is a multiplicative $(1 \pm (\delta_1 + \delta_2 + \delta_1\delta_2 + 2^{-\alpha}))$-approximation to X_1X_2. If $\delta_1 < 1$, then the product is a $(1 \pm (\delta_1 + 2\delta_2 + 2^{-\alpha}))$-approximation.

For each i of interest, one can easily compute a multiplicative $(1\pm C\cdot 2^{-\alpha})$-approximation for $\frac{k+1-i}{i}\cdot\frac{a}{b-a}$ in our arithmetic, where $C > 1$ is a constant. We make the assumption that $3Ck2^{\alpha} \leq 1$, which we satisfy later by setting a sufficiently large α. Hence we use Equation 31 to obtain a sequence of multiplicative $(1 \pm 3Ck2^{-\alpha})$-approximations t'_i for t_i, where $0 \leq i \leq s$. At the end, we find the maximum t'_i, which is represented as a pair $\langle S_i, E_i\rangle$. For all other t'_i, we no longer require that $S_i \geq 2^{\alpha}$ and we modify their representation $\langle S_i, E_i\rangle$ so that E_i is the same as in the representation of the maximum t'_i. In the process we may lose least significant bits of the some t'_i or even all non-zero bits. Assuming again that $3Ck2^{-\alpha} < 1$, the maximum additive error $|t_i - t'_i|$ we get for each i for the modified representation is bounded by

$$3Ck2^{-\alpha}\cdot t_i + 2^{-\alpha}\cdot\max_j t'_j \leq 3Ck2^{-\alpha}\cdot t_i + 2\cdot 2^{-\alpha}\cdot\max_j t_j \leq (3Ck+2)\cdot 2^{-\alpha}\cdot\max_j t_j,$$

where the first error term comes from the multiplicative error we obtain approximating each t_i and the second error term comes from making all exponents in the representation match the exponent of the largest t'_i. Finally, we set $\alpha = \lceil\log((3Ck+2)\cdot 4(k+1)^2Q)\rceil$. This meets the previous assumption that $3Ck2^{-\alpha} < 1$ and the guarantee on the error we may make on each t'_i is as desired. Note that since k and Q can be represented using a constant number of words, so can integers of size at most $2^{\alpha+1}$. To bound β, observe that every $\frac{k+1-i}{i}\cdot\frac{a}{b-a}$ lies in the range $[1/kb, kb]$, which implies that all t_i lie in $[1/kb^k, kb^k]$, and the maximum absolute value of an exponent we need is of order $O(k\log(kb))$, which can be stored using a constant number of machine words.

To generate a random number from $\mathcal{D}$, we consider only the significands S_i in the final modified representation of t'_i's, and select each i with probability $S_i/\sum_{j<s}S_j = t'_i/\sum_{j<s}t'_j$. The total running time of the algorithm is $O(s)$. ■

We are ready to prove that the entire algorithm can be implemented efficiently. We use the algorithm of Lemma 4.11 for efficiently simulating Step 3 in the operation `lowest`.

Theorem 4.13 *Consider an algorithm $\mathcal{A}$ that queries the input graph only via the oracle described as Algorithm 1. Let $t \geq 1$ be a bound on the expected resulting number of calls in $\mathcal{A}$ to the oracles described as Algorithm 1 and Algorithm 2, and such that t fits into a constant number of machine words using the standard binary representation. Let d be an upper bound on the maximum degree of the input graph.*

Suppose that calls to Algorithm 1 are replaced with calls to Algorithm 3. The oracles described as Algorithm 3 and Algorithm 4 can be implemented in such a way that with probability $4/5$ all of the following events hold:

- *The number of queries to the graph is $O(t\cdot\log^2(dt))$.*
- *The total time necessary to compute the answers for the queries to the oracles is $O(t\cdot\log^3(dt))$.*
- *The distribution of the answers that the oracle gives is $\mathcal{D}$ such that for some other distribution $\mathcal{D}'$ over answers, the convex combination $\frac{4}{5}\cdot\mathcal{D} + \frac{1}{5}\cdot\mathcal{D}'$ is the distribution of answers of the oracle described as Algorithm 1.*

Proof: Let $a_\star = d \cdot O(t)$, where $O(t)$ is the bound from Lemma 4.8 on the number of vertices for which the operation `lowest` is called. If the event specified in Lemma 4.8 occurs, then $a_\star$ is an upper bound on the number of edges for which the process for generating random numbers is simulated. Let $b_\star = O(t) \cdot (d_\star + 1) = O(t \log d)$, where $O(t)$ is the same bound as above. Then $b_\star$ bounds the number of times Step 3 in Algorithm 9 is run, provided the event specified in Lemma 4.8 occurs. Let $Q = 20b_\star$.

Let $c_\star = \max\{d_\star, \lceil \log(20a_\star^2) \rceil\}$. Since it is impossible to generate and store real numbers, we assign to edges uniform random numbers from the set $\{i/2^{c_\star} : 1 \leq i \leq 2^{c_\star}\}$, instead of the set $(0, 1]$. This can be seen as selecting a random number from $(0, 1]$ and then rounding it up to the next multiplicity of $1/2^{c_\star}$. In particular, for every $i \in \{1, \ldots, 2^{c_\star}\}$, all numbers in $((i-1)/2^{c_\star}, i/2^{c_\star}]$ become $i/2^{c_\star}$. Observe also that each range $\mathcal{I}_j$ is a union of some number of sets $((i-1)/2^{c_\star}, i/2^{c_\star}]$, because $c_\star \geq d_\star$. This means that there is no need to modify the process for generating random numbers, except for selecting a random $i/2^{c_\star}$ in a specific $\mathcal{I}_j$, instead of an arbitrary real number from $\mathcal{I}_j$. Observe also that as long we do not select the same number $i/2^{c_\star}$ twice, the entire exploration procedure behaves in the same way as in the idealized algorithm, since the ordering of numbers remains the same.

Note that due to the assumption in the lemma statement, t can be represented in the standard binary form, using a constant number of machine words. This is also the case for d, because of the standard assumption that we can address all neighbors of all vertices in neighbor queries. This implies that $Q = O(t \log d)$ also has this property. Finally, the probabilities $\frac{\texttt{next_lb-lb}}{\texttt{1-lb}}$ can easily be expressed using fractions a/b, where a and b are of order $O(d)$, and therefore, fit into a constant number of machine words as well. This implies that we can use the algorithm of Lemma 4.11. Instead of directly simulating Step 3, we proceed as follows. First, we run the algorithm of Lemma 4.11 with the error parameter Q to select a number t of edges in T. Then we select a random subset of edges of size t. This can be done in $O(t \log d)$ time.

We show that the algorithms and data structures can be implemented in such a way that the main claim of the theorem holds, provided the following events occur:

- the events described in the statement of Lemma 4.8,
- the rounded numbers assigned to the first $a_\star$ edges for which the process for generating random numbers is simulated are different,
- the first $b_\star$ simulations of the algorithm described by Lemma 4.11 do not result in selecting a random number from the part on which the output distribution of the algorithm and the binomial distribution differ.

The first of the events does not happen with probability at most 1/10. This follows from Lemma 4.8. Consider the second event. The probability that two random numbers $i/2^{c_\star}$ are identical is bounded by $1/2^{c_\star} \leq 1/(20a_\star^2)$. Consider the first $a_\star$ edges for which the process generating random numbers is run. The expected number of pairs of the edges that have the same random number is bounded by $a_\star^2 \cdot 1/(20a_\star^2) = 1/20$. By Markov's inequality, the probability that two of the edges have the same random number assigned is bounded by $1/20$. Finally, the probability that the last event does not occur is bounded by $1/20$ as well via the union bound. Summarizing, the events occur with probability at least $4/5$.

We now bound the running time, provided the above events occur. We assume that we use a standard data structure (say, balanced binary search trees) to maintain collections of items. The time necessary for each operation in these data structures is of order at most the logarithm of the maximum collection size. For instance, we keep a collection of data structures `neighbors`$[v]$ for v that appear in our algorithm. We create `neighbors`$[v]$ for a given v only when it is accessed for the first time. Observe that the number of

v for which we have to create `neighbors`$[v]$ is bounded by the query complexity $O(t \log^2(dt))$, because of how we access vertices. Therefore, accessing each `neighbors`$[v]$ requires at most $O(\tau)$ time, where we write τ to denote the logarithm of the bound on the query complexity. That is, $\tau = O(\log t + \log\log d)$.

The time necessary to run Algorithm 3 is bounded by $O(\tau)$, which we need to locate the data structure `neighbors`$[v]$ for a given v, plus $O(1)$ time per each call to Algorithm 4 (we do not include the cost of running Algorithm 4 or the operation `lowest` here; they are analyzed later). The amount of computation in Algorithm 3 without the resulting calls to other procedures is bounded by $O(t \cdot \tau)$.

Consider now Algorithm 4. In every run, we first spend $O(\log t)$ time to check if we have already computed the answer for a given edge. Then locating the data structures `neighbors`$[u]$ and `neighbors`$[v]$ for the endpoints u and v costs at most $O(\tau)$. The running time of the reminder of the algorithm requires time proportional to the number of recursive calls. Therefore, the total amount of time spent executing Algorithm 4 (without calls to other procedures) is bounded by $O(t \cdot \tau)$.

We now bound the running time necessary to execute all operations of data structures `neighbors`. The initialization of `neighbors`$[v]$ (Algorithm 6) for a given v can be done $O(1)$ time plus $O(\tau)$ time necessary for inserting the data structure into the collection of all `neighbors`$[v]$. Overall, since at most $O(t \log^2(dt))$ data structures are created, the total time necessary to initialize the data structures `neighbors`$[v]$ is $O(t \cdot \log^2(dt) \cdot \tau)$. Setting a value for some edge in Algorithm 7 takes at most $O(\log d)$ time to insert the value into the mapping. This operation is run at most once for every neighbor query, so the total amount of computation in this procedure is $O(t \cdot \log^2(dt) \cdot \log d)$. So far, the total computation time is bounded by $O(t \log^3(dt))$.

Clearly, running the operation described by Algorithm 8 takes $O(1)$ time, so overall the total amount of computation in all executions of Algorithm 8 is not greater than some constant times the total amount of computation in the operation `lowest` (Algorithm 9). Hence it suffices to bound the total amount of computation in Algorithm 9, which we do next.

Recall that Algorithm 9 is run at most $O(t)$ times. Therefore all operations in the loop **while** are run at most $O(t \log d)$ times. The total size of sets S in Step 2 is bounded by the query complexity, and discovering each element of S costs at most $O(\log d)$ time, if the data structure assigned_number is properly implemented, using augmented balanced binary search trees. Therefore the total cost of running Step 2 is at most $O(t \cdot \log d + t \cdot \log^2(dt) \cdot \log d) = O(t \cdot \log^2(dt) \cdot \log d)$. In Step 3, we use the algorithm of Lemma 4.11. The total toll for running the algorithm is $O(t \cdot \log(dt))$. Therefore, the total time necessary to simulate all executions of Step 2 is bounded by $O((t \cdot \log d + t \cdot \log(dt)) \cdot \log Q) = O(t \cdot \log^2(dt))$. The total number of executions of the body of the loop **foreach** in Step 4 is bounded by the query complexity $O(t \cdot \log^2(dt))$ times 2. The time required to execute the body of the loop is dominated by the following two kinds of operations. One kind is querying and modifying the data structure `assigned_number`$[w]$ and the data structure for S. With a proper implementation (say, augmented balanced binary search trees) these operations take at most $O(\log d)$ time each. The other kind of operation is locating `neighbors`$[w]$ for the discovered neighbor w, which takes most $O(\tau)$ time. The total computation time for all executions of the loop **foreach** is therefore bounded by $O(t \cdot \log^3(dt))$.

Finally sorting S never takes more than $O(|S| \log d)$ time, because $|S| \leq d$, and each element of S can be added at the end of the list `sorted` in amortized $O(1)$ time if the list is implemented using extendable arrays. This amounts to $O(t \cdot \log^2(dt) \cdot \log d)$ in all executions of Step 11. At the end of the operation, the requested k^{th} adjacent edge can be returned in $O(1)$ time.

Summarizing, the computation of the answers of the oracles takes at most $O(t \cdot \log^3(dt))$ time, if all the desired events occur, which happens with probability at least $4/5$. Note that when these events occur,

then also despite rounding random numbers assigned to edges, the implementation does not diverge from the behavior of the idealized oracle. ■

5 The Near-Optimal Algorithms

Theorem 3.1 gives a bound on the expected number of recursive calls to oracles, sufficient to compute an answer when the vertex cover oracle is called for a random vertex. The expected number of calls is $O(\rho \cdot \bar{d})$, where ρ is the ratio between the maximum degree d and the minimum degree $d_{\min}$, and $\bar{d}$ is the average degree. (Recall that we assume without loss of generality that $d_{\min} \geq 1$. For isolated vertices, the oracle answers that they are not in the vertex cover in $O(1)$ time, and therefore, it suffices to focus on the subgraph consisting of non-isolated vertices.)

A straightforward application of Theorem 3.1 gives a bound of $O(d^2)$ for graphs with maximum degree bounded by d. We show a bound of $O(d/\epsilon)$ for a *modified* graph, which is preferable if $1/\epsilon < d$, and we also show how to use the modified graph to obtain an estimate for the minimum vertex cover size in the original input graph. We combine the obtained bound with Theorem 4.13 to get a fast and query-efficient algorithm.

Next we show how to obtain an efficient algorithm for the case when only the average degree of the input graph is bounded. Finally, we show how to adapt the algorithm to the dense graph case, when only vertex-pair queries are allowed.

5.1 Bounded Maximum Degree

As we have mentioned above, we can assume that $1/\epsilon < d$. We transform our graph into one with large minimum degree, so that the ratio of maximum to minimum degree is small. For a given graph $G = (V, E)$ with maximum degree d, consider a graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, such that $\widetilde{V} = V \cup V'$ and $\widetilde{E} = E \cup E'$ where V' and E' are defined as follows. The set V' contains a "shadow" vertex v' for each vertex $v \in V$, and E' contains $\lfloor \epsilon d \rfloor$ parallel edges between v and v', and $8d$ parallel self-loops for v'.

For a random ranking $\tilde{\pi}$ over $\widetilde{E}$, for the output vertex cover $\mathrm{C}^{\tilde{\pi}}(\widetilde{G})$ on the new graph $\widetilde{G}$, we are interested in bounding the size of $\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V$ as compared to $\mathrm{VC}_{\mathrm{opt}}(G)$ (the size of a minimum vertex cover of G). Since $\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V$ is a vertex cover of G, we have that $|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V| \geq \mathrm{VC}_{\mathrm{opt}}(G)$, and so we focus on an upper bound for $|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V|$.

Let $\tilde{F}$ be the set of all parallel edges connecting each v with the corresponding v'. By the properties of the construction of $\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V$, we have

$$|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V| \leq 2|\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap E| + |\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}| \leq 2\mathrm{VC}_{\mathrm{opt}}(G) + |\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}|.$$

Consider an arbitrary ranking $\tilde{\pi}$ of $\tilde{E}$. Observe that for each $v \in V$, the matching $\mathrm{M}^{\tilde{\pi}}(\widetilde{G})$ either includes a parallel edge between v and v' or it includes a self-loop incident to v'. For every $v' \in V'$, if the lowest rank of self-loops incident to v' is lower than the lowest rank of edges (v, v'), then $\mathrm{M}^{\tilde{\pi}}(\widetilde{G})$ contains one of the self-loops, and does not contain any parallel edge (v, v'). If the ranking $\tilde{\pi}$ is selected uniformly at random, the above inequality on ranks does not hold for each vertex independently with probability at most $\epsilon d/8d = \epsilon/8$. Therefore, the expected number of edges in $\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}$ is upper bounded by $\epsilon n/8$. Without loss of generality, we can assume that $\epsilon n > 72$, since otherwise we can read the entire input with only $O(1/\epsilon^2)$ queries and compute a maximal matching in it. It follows from the Chernoff bound that with probability $1 - 1/20$, $|\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}| \leq \epsilon n/4$.

Observe that given query access to G, we can provide query access to $\widetilde{G}$ (in particular, the edges in E' that are incident to each $v \in V$ can be indexed starting from $\deg(v)+1$). Every query to $\widetilde{G}$ can be answered in $O(1)$ time, using $O(1)$ queries to G. Therefore, we can simulate an execution of the vertex-cover and the maximal-matching oracles on $\widetilde{G}$.

Note that the expected number of recursive calls to the maximal matching oracle is bounded for a random vertex $v \in \tilde{V}$ by $O(d/\epsilon)$, because the maximum degree and the minimum degree are within a factor of $O(1/\epsilon)$. Also note that since $|V| = |\tilde{V}|/2$, this expectation for a random vertex $v \in V$ is at most twice as much, i.e., it is still $O(d/\epsilon)$.

For any ranking $\tilde{\pi}$ of edges in $\widetilde{E}$, if we sample $O(1/\epsilon^2)$ vertices from V with replacement, then the fraction of those in $\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V$ is within an additive $\epsilon/8$ of $|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V|/|V|$ with probability at least $1 - 1/20$. Let μ be this fraction of vertices. Therefore, we have that

$$\mathrm{VC}_{\mathrm{opt}}(G) - \epsilon n/4 \;\leq\; \mu \cdot n \;\leq\; 2\mathrm{VC}_{\mathrm{opt}}(G) + \epsilon n/2$$

with probability at least $1 - 1/10$. Thus $(\mu + \epsilon/4) \cdot n$ is the desired $(2, \epsilon n)$-estimate. The expected number of calls to the vertex cover and maximal matching oracles is bounded by $O(d/\epsilon^3)$. Note that without loss of generality, $\epsilon \geq 1/4n$, because any additive approximation to within an additive factor smaller than $1/4$ yields in fact the exact value. Therefore the expected number of calls to the oracles is bounded by $O(n^4)$, which can be represented with a constant number of machine words in the standard binary representation, using the usual assumption that we can address all vertices of the input graph. By applying now Theorem 4.13, we obtain an implementation of the algorithm. It runs in $O(d/\epsilon^3 \cdot \log^3(d/\epsilon))$ time and makes $O(d/\epsilon^3 \cdot \log^2(d/\epsilon))$ queries. Moreover, the probability that the implementation diverges from the ideal algorithm is bounded by $1/5$. Therefore, the implementation outputs a $(2, \epsilon n)$-estimate with probability $1 - 1/10 - 1/5 \geq 2/3$.

Corollary 5.1 *There is an algorithm that makes $O(\frac{d}{\epsilon^3} \cdot \log^3 \frac{d}{\epsilon})$ neighbor and degree queries, runs in $O(\frac{d}{\epsilon^3} \cdot \log^3 \frac{d}{\epsilon})$ time, and with probability $2/3$, outputs a $(2, \epsilon n)$-estimate to the minimum vertex cover size.*

5.2 Bounded Average Degree

In this section, we assume an upper bound $\bar{d}$ on the average graph degree and show an efficient algorithm in this case.[6] To do this, we will transform the graph into a new graph for which the ratio of the maximum degree to the minimum degree is small.

Our first transformation is to automatically add high degree vertices to the cover, and continue by finding a cover for the graph that is induced by the remaining vertices. Given a graph $G = (V, E)$ with average degree $\bar{d}$, let L denote the subset of vertices in G whose degree is greater than $8\bar{d}/\epsilon$. Hence, $|L| \leq \epsilon n/8$. Let $E(L)$ denote the subset of edges in G that are incident to vertices in L, and let $\overline{G} = (\overline{V}, \overline{E})$ be defined by $\overline{V} = V \setminus L$ and $\overline{E} = E \setminus E(L)$, so that the maximum degree in $\overline{G}$ is at most $8\bar{d}/\epsilon$. For any maximal matching $\overline{M}$ in $\overline{G}$ we have that

$$\mathrm{VC}_{\mathrm{opt}}(G) \;\leq\; 2|\overline{M}| + |L| \;\leq\; 2\mathrm{VC}_{\mathrm{opt}}(G) + \frac{\epsilon}{8} n \,.$$

Thus, the first modification we make to the oracles is that if the vertex-cover oracle is called on a vertex v such that the degree of v is greater than $(4/\epsilon)\bar{d}$, then it immediately returns TRUE.

[6] As shown in [PR07], we don't actually need to know $\bar{d}$ for this purpose, but it suffices to get a bound that is not much higher than $(4/\epsilon)\bar{d}$, and such that the number of vertices with a larger degree is $O(\epsilon n)$, where such a bound can be obtained efficiently.

The remaining problem is that when we remove the high degree vertices, there are still edges incident to vertices with degree at most $(4/\epsilon)\bar{d}$ whose other endpoint is a high degree vertex, and this is not known until the appropriate neighbor query is performed. We deal with this by adding shadow vertices to replace the removed high degree vertices. At the same time, we increase the minimum degree as in the previous subsection. We now create a graph $\widetilde{G} = (\overline{V} \cup \widetilde{V}, \overline{E} \cup \widetilde{E})$ as follows. For every $v \in \overline{V}$, we add to $\widetilde{V}$ a vertex v' and vertices v''_i, where $1 \leq i \leq \lceil \deg_G(v)/\bar{d} \rceil$ and $\deg_G(v)$ is the degree of v in G. Each of these new vertices has $32\bar{d}/\epsilon$ parallel self-loops. Moreover, we add $\bar{d}$ parallel edges between v and v'. Finally, partition the edges incident to v in G into $\lceil \deg_G(v)/\bar{d} \rceil$ groups, each of size at most $\bar{d}$. The first group corresponds to the first $\bar{d}$ edges on the neighborhood list of v, the second group corresponds to the next $\bar{d}$ edges, and so on. Let $E_{v,i} \subset E$ be the set of edges in the i-th group. For every i of interest, we add $|E_{v,i} \cap E(L)|$ parallel edges between v and v''_i and $|E_{v,i} \setminus E(L)|$ parallel self-loops incident to v''_i. We add these edges so that we are later able to simulate every query to $\widetilde{G}$ using a constant number of queries to G.

Let us bound the total number of vertices in $\widetilde{V}$. The number of vertices v' is $|\overline{V}|$. The number of vertices v''_i is bounded by

$$\sum_{v \in \overline{V}} \left\lceil \frac{\deg_G(v)}{\bar{d}} \right\rceil \leq \sum_{v \in \overline{V}} \left(\frac{\deg_G(v)}{\bar{d}} + 1 \right) \leq \frac{\bar{d} \cdot |\overline{V}|}{\bar{d}} + |\overline{V}| = 2|\overline{V}|,$$

because $\widetilde{V}$ has been created by removing vertices with highest degrees in G, and the average degree of vertices in $\widetilde{V}$ in G cannot be greater than $\bar{d}$, the initial average degree. This shows that $|\widetilde{V}| \leq 3|\overline{V}|$.

We now repeat an argument from the previous section that despite the additional edges and vertices, $|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap \widetilde{V}|$ is likely to be a good approximation to $\mathrm{VC}_{\mathrm{opt}}(\overline{G})$ for a random ranking $\tilde{\pi}$. First, $\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap \widetilde{V}$ is still a vertex cover for $\overline{G}$, so $\mathrm{VC}_{\mathrm{opt}}(\overline{G}) \leq |\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap \widetilde{V}|$. Let $\tilde{F}$ be the set of edges connecting all v with the corresponding v' and v''_i. We have

$$|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap \overline{V}| \leq 2|\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \overline{E}| + |\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}| \leq 2\mathrm{VC}_{\mathrm{opt}}(\overline{G}) + |\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}|.$$

Observe that if for some of the vertices in $\widetilde{V}$, the lowest rank of self-loops is lower than the lowest rank of the parallel edges connecting this vertex to the corresponding vertex in $\overline{V}$, then one of the self-loops is selected for the maximal matching as opposed to the parallel edges. The inequality on ranks does not hold with probability at most $\bar{d}/(32\bar{d}/\epsilon) = \epsilon/32$ independently for each vertex in $\widetilde{V}$. It therefore follows from the Chernoff bound that the number of edges in $\mathrm{M}^{\tilde{\pi}}(\widetilde{G}) \cap \tilde{F}$ is not bounded by $\epsilon|\widetilde{V}|/16$ with probability at most $\exp(-\epsilon|\widetilde{V}|/32)$, which is less than $1/20$ if $|\widetilde{V}| > 100/\epsilon$, and we can assume that this is the case. (To circumvent the case of $|\widetilde{V}| \leq 100/\epsilon$, we can modify the algorithm as follows. If a sampled vertex belongs to a connected component in $\overline{V}$ of size at most $100/\epsilon$, then we can read its connected component in $\widetilde{G}$ and deterministically find a maximal matching that uses only edges in $\overline{E}$ and self-loops in $\widetilde{E}$. This all takes at most $O(\bar{d}/\epsilon^2)$ time, which as we see later, we are allowed to spend per each sampled vertex.) Therefore, we have

$$|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap \overline{V}| \leq 2\mathrm{VC}_{\mathrm{opt}}(\overline{G}) + \epsilon|\overline{V}|/4,$$

with probability at least $1 - 1/20$.

Observe that given query access to G, we can efficiently provide query access to $\widetilde{G}$. Degrees of vertices in $\overline{V}$ are the same as in G. For associated vertices in $\widetilde{V}$ it is easy to compute their degree in $O(1)$ time, using the degree of the corresponding vertex in $\overline{V}$. To answer neighbor queries for vertices v in $\overline{V}$, except for the fixed connections to v', it suffices to notice that if the corresponding edge in G is connected to a vertex in L, this connection is replaced by a connection to an appropriate vertex v''_i. Otherwise, the edge remains in $\overline{E}$.

For vertices v_i'' some number of connections can either be a connection to the corresponding v or a self-loop. This can be checked in $O(1)$ time with a single query to the neighborhood list of v. All the other edges are fixed. Therefore, we can simulate an execution of the vertex-cover and the maximal-matching oracles on $\widetilde{G}$. Answering every query to $\widetilde{G}$ requires $O(1)$ time and $O(1)$ queries to G. Sampling vertices uniformly at random from in $\overline{V} \cup \widetilde{V}$ is more involved, but in our algorithm, we only need to sample vertices from V, which we assume we can do.

The expected number of recursive calls to the maximal matching oracle is bounded by $O(\bar{d}/\epsilon^2)$ for a random vertex $v \in \overline{V} \cup \widetilde{V}$, because the maximum degree and the minimum degree are within a factor of $O(1/\epsilon)$ and the maximum degree is bounded by $O(\bar{d}/\epsilon)$. Note that since $3|\overline{V}| \geq |\widetilde{V}|$, this expectation for a random vertex $v \in V$ is at most twice as much, i.e., it is still $O(\bar{d}/\epsilon^2)$.

For any ranking $\tilde{\pi}$ of edges in $\widetilde{G}$, if we sample $O(1/\epsilon^2)$ vertices from V with replacement, then the fraction of those for which the oracle answers TRUE is within an additive error $\epsilon/8$ of the total fraction of vertices for which the oracle answers TRUE. with probability $1 - 1/20$. Let μ be the fraction of sampled vertices. We have

$$\mathrm{VC}_{\mathrm{opt}}(G) - \epsilon n/8 \leq \mu \cdot n \leq 2\mathrm{VC}_{\mathrm{opt}}(G) + \epsilon n/4 + \epsilon n/8 + \epsilon n/8$$

with probability $1 - 1/10$. Then $(\mu + \epsilon/8)n$ is the desired $(2, \epsilon n)$-estimate. The expected number of calls to the vertex cover and maximal matching oracles is bounded by $O(d/\epsilon^4)$. As before, without loss of generality, this quantity can be bounded by $O(n^5)$, which fits into a constant number of machine words. By applying now Theorem 4.13, we obtain an implementation of the algorithm. It runs in $O(d/\epsilon^3 \cdot \log^3(d/\epsilon))$ time and makes $O(d/\epsilon^3 \cdot \log^2(d/\epsilon))$ queries. Moreover, the probability that the implementation diverges from the ideal algorithm is bounded by $1/5$. Therefore, the implementation outputs a $(2, \epsilon n)$-estimate with probability at least $1 - 1/5 - 1/10 \geq 2/3$.

Corollary 5.2 *There is an algorithm that makes $O(\frac{\bar{d}}{\epsilon^4} \cdot \log^2 \frac{\bar{d}}{\epsilon})$ neighbor and degree queries, runs in $O(\frac{\bar{d}}{\epsilon^4} \cdot \log^3 \frac{\bar{d}}{\epsilon})$ time, and with probability $2/3$, outputs a $(2, \epsilon n)$-estimate to the minimum vertex cover size.*

5.3 Adapting the Algorithm to the Vertex-Pair Query Model

The focus of this paper was on designing a sublinear-time algorithm whose access to the graph is via degree queries and neighbor queries. In other words, we assumed that the graph was represented by adjacency lists (of known lengths). When a graph is dense (i.e., when the number of edges is $\Theta(n^2)$), then a natural alternative representation is by an adjacency matrix. This representation supports queries of the form: "Is there an edge between vertex u and vertex v?", which we refer to as *vertex-pair* queries.

We next show how to adapt the algorithm described in the previous section to an algorithm that performs vertex-pair queries. The query complexity and running time of the algorithm are (with high constant probability) $\tilde{O}(n/\epsilon^4)$, which is linear in the average degree for dense graphs. As in the previous section, the algorithm outputs (with high constant probability) a $(2, \epsilon)$-estimate of the size of the minimum vertex cover. We recall that the linear lower bound in the average degree [PR07] also holds for the case that the average degree is $\Theta(n)$ and when vertex-pair queries are allowed.

Given a graph $G = (V, E)$, let $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ be a supergraph of G that is defined as follows. For every vertex $v \in V$ whose degree in G is less than[7] n, there exists a vertex $v' \in \widetilde{V}$, where there are

[7] If there are no self-loops in the original graph, then the bound is $n - 1$.

$n - \deg_G(v)$ parallel edges between v and v', and there are $(8/\epsilon)n$ self-loops incident to v'. As shown in Subsection 5.1, with high probability over the choice of a ranking $\tilde{\pi}$ over $\widetilde{G}$, we have that $|\mathrm{C}^{\tilde{\pi}}(\widetilde{G}) \cap V| \leq 2\mathrm{VC}_{\mathrm{opt}}(G) + (\epsilon/4)n$.

Note that we can emulate neighbor queries to $\widetilde{G}$ given access to vertex-pair queries in G as follows. Let the vertices in G be $\{1, \ldots, n\}$. When the j^{th} neighbor of vertex i is queried, then the answer to the query is j when $(i, j) \in E$, and it is i' (the new auxiliary vertex adjacent to i) when $(i, j) \notin E$. The degree of every vertex in V is n, so there is no need to perform degree queries. Since the ratio between the maximum degree and the minimum degree in $\widetilde{G}$ is at most $1/\epsilon$, and the maximum and average degrees are $O(n/\epsilon)$, we obtain an algorithm whose complexity is $\tilde{O}(n/\epsilon^4)$, as claimed.

References

[BSS08] Itai Benjamini, Oded Schramm, and Asaf Shapira. Every minor-closed property of sparse graphs is testable. In *STOC*, pages 393–402, 2008.

[CEF+05] Artut Czumaj, Funda Ergun, Lance Fortnow, Avner Magen, Ilan Newman, Ronitt Rubinfeld, and Christian Sohler. Approximating the weight of the euclidean minimum spanning tree in sublinear time. *SIAM Journal on Computing*, 35(1):91–109, 2005.

[CHW08] Andrzej Czygrinow, Michal Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *DISC*, pages 78–92, 2008.

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

[CS09] Artur Czumaj and Christian Sohler. Estimating the weight of metric minimum spanning trees in sublinear time. *SIAM Journal on Computing*, 39(3):904–922, 2009.

[CSS09] Artur Czumaj, Asaf Shapira, and Christian Sohler. Testing hereditary properties of nonexpanding bounded-degree graphs. *SIAM J. Comput.*, 38(6):2499–2510, 2009.

[Ele10] Gábor Elek. Parameter testing in bounded degree graphs of subexponential growth. *Random Struct. Algorithms*, 37(2):248–270, 2010.

[Fei06] Uriel Feige. On sums of independent random variables with unbounded variance, and estimating the average degree in a graph. *SIAM Journal on Computing*, 35(4):964–984, 2006.

[GR08] Oded Goldreich and Dana Ron. Approximating average parameters of graphs. *Random Structures and Algorithms*, 32(4):473–493, 2008.

[GRS10] Mira Gonen, Dana Ron, and Yuval Shavitt. Counting stars and other small subgraphs in sublinear time. In *SODA*, pages 99–116, 2010.

[HKNO09] Avinatan Hassidim, Jonathan A. Kelner, Huy N. Nguyen, and Krzysztof Onak. Local graph partitions for approximation and testing. In *FOCS*, pages 22–31, 2009.

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

[NO08] Huy N. Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *FOCS*, pages 327–336, 2008.

[NS11] Ilan Newman and Christian Sohler. Every property of hyperfinite graphs is testable. In *STOC*, 2011. To appear.

[PR07] Michal Parnas and Dana Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theor. Comput. Sci.*, 381(1-3):183–196, 2007.

[PS98] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover publications, 1998.

[Yos11] Yuichi Yoshida. Optimal constant-time approximation algorithms and (unconditional) inapproximability results for every bounded-degree CSP. In *STOC*, 2011.

[YYI09] Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. An improved constant-time approximation algorithm for maximum matchings. In *STOC*, pages 225–234, 2009.