Property Testing in Bounded Degree Graphs

Oded Goldreich* Dana Ron†

February 18, 1997

Abstract

We further develop the study of testing graph properties as initiated by Goldreich, Goldwasser and Ron. Whereas they view graphs as represented by their adjacency matrix and measure distance between graphs as a fraction of all possible vertex pairs, we view graphs as represented by bounded-length incidence lists and measure distance between graphs as a fraction of the maximum possible number of edges. Thus, while the previous model is most appropriate for the study of dense graphs, our model is most appropriate for the study of bounded-degree graphs.

In particular, we present randomized algorithms for testing whether an unknown bounded-degree graph is connected, k-connected (for $k > 1$), planar, etc. Our algorithms work in time polynomial in $1/\epsilon$, always accept the graph when it has the tested property, and reject with high probability if the graph is ϵ-away from having the property. For example, the 2-Connectivity algorithm rejects (w.h.p.) any N-vertex d-degree graph for which more than ϵdN edges need to be added to make the graph 2-edge-connected.

In addition we prove lower bounds of $\Omega(\sqrt{N})$ on the query complexity of testing algorithms for the Bipartite and Expander properties.

KEYWORDS: Approximation Algorithms, Graph Algorithms.

*Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, ISRAEL. E-mail: oded@wisdom.weizmann.ac.il. On sabbatical leave at LCS, MIT.

†Laboratory for Computer Science, MIT, 545 Technology Sq., Cambridge, MA 02139. E-mail: danar@theory.lcs.mit.edu. Supported by an NSF postdoctoral fellowship.

1 Introduction

Approximation is one of the basic paradigms of modern science. One of its facets in computer science is approximation algorithms. Yet, it is not always clear what approximation means. The dominant approach considers a cost function associated with possible solutions of an instance, and regards an approximation algorithm as one which provides an *approximation of the cost of an optimal solution.* In many cases one also expects (or requires) the approximation algorithm to supply a solution with cost close to optimal. This approach is most suitable in case there is a natural cost measure for candidate solutions and the optimal solution is preferable only due to its low(est) cost. An alternative approach is to consider the *distance* of the given instance *to the closest instance which has a desirable property.* The property may be having a solution of certain cost (w.r.t some cost measure defined as in the first approach), but it can also be of a qualitative nature; for example, being a connected graph (in case the instances are graphs), or being a linear function (in case the instances are functions). The latter approach underlines all work on testing low-degree polynomials [BLR93, RS96, GLR+91, BFL91, BFLS91, FGL+91, ALM+92] and codes [BFLS91, ALM+92, BGS95, Hås96], and its relevance to the construction of probabilistically checkable proofs [BFL91, BFLS91, FGL+91, AS92, ALM+92] is well known. Recently, a general formulation of property testing has been presented in [GGR96], and its connection to the former approach to approximation have been demonstrated. Still the two approaches do differ, and the question of meaningfulness has to be addressed (as we do below).

Another general point is that approximation is applicable not only when the optimization problems are intractable. Also in case there exists an efficient algorithm for solving the problem optimally, one may wish to have an even faster algorithm and be willing to tolerate its approximative nature. In particular, in a RAM model of computation, an approximation algorithm may even run in sub-linear time and still provide valuable information. For example, the testing algorithms of [GGR96] run in constant time and provide "constant error approximations" (e.g., one can approximate the value of the maximum cut in a dense graph to within a constant factor in constant time).

1.1 Testing graph properties

Recently, a study of testing graph properties was initiated by Goldreich *et. al.*, as part of a general study of *Property Testing* [GGR96]. In the general model, the algorithm is given oracle access[1] to a function and has to decide whether the function has some specified property or is "far" from having that property. Distance between functions is defined as the fraction of instances on which the functions' values differ.[2] In their study of *testing graph properties*, Goldreich *et. al.* view the graph as a Boolean function defined over the set of all vertex-pairs. Thus, their measure of distance between graphs is the fraction of vertex-pairs which are an edge in one graph and a non-edge in the other graph, taken over the total number of vertex-pairs. This model is most appropriate for the study of dense graphs, and indeed the graph algorithms in [GGR96] refer mainly to dense graphs. For example, their (constant time) Monte Carlo algorithm for testing whether a graph is Bipartite or is 0.1-far from Bipartite is meaningful only for N-vertex graphs which have more than $0.1 \cdot \binom{N}{2}$ edges (as any graph having fewer edges is 0.1-close to being Bipartite). Furthermore, testing connectivity in this model is trivial as long as the distance parameter is bigger than $\frac{2}{N}$ (since

[1] Here we ignore the variant in which the algorithm is given only random examples.

[2] We ignore the variant where distance is measured with respect to an arbitrary distribution (rather than w.r.t the uniform one).

every N-vertex graph is $\frac{2}{N}$-close to being connected and so the algorithm may as well accept any graph).

In this paper we present an alternative model. We view bounded-degree graphs as functions defined over pairs (v, i), where v is a vertex and i is a positive integer within a predetermined (degree) bound, denoted d. The range of the function is the vertex set augmented by a special symbol. Thus the value on argument (v, i) specifies the i^{th} neighbor of v (with the special symbol indicating non-existence of such a neighbor). Our measure of distance between (N-vertex) graphs is the fraction of vertex-pairs which are an edge in one graph and a non-edge in the other, taken over the size of the domain (i.e., over dN). Thinking of d as being a fixed constant, this model does not allow to consider dense graphs, yet it is most appropriate to the study of graphs with maximum degree d. In particular, it is no longer true that every (degree-d) graph is 0.1-close to being connected and so an algorithm for testing connectivity cannot be trivial (i.e., always accept). On the other hand, the techniques in [GGR96] do not apply to our model and the analogies of most of the results in [GGR96] do not hold: For example, we show that no constant time (Monte Carlo) algorithm can test whether a graph is Bipartite or is 0.1-far from Bipartite, where distance is as defined here.

To demonstrate the viability of our model, we present randomized algorithms for testing several natural properties of bounded-degree graphs. All algorithms get as input a degree bound d and an approximation parameter ϵ. The algorithms make queries of the form (v, i) which are answered with the name of the i^{th} neighbor of v (or with a special symbol in case v has less than i neighbors). With probability at least $2/3$, each algorithm accepts any graph having the tested property and rejects any graph which is at distance greater than ϵ from any graph having the property. Actually, except for the cycle-freeness tester, all algorithms have one-sided error (i.e., always accept graphs which have the property), and furthermore when rejecting they present a short certificate vouching that the property does not hold in the tested graph. Assuming that vertex names are manipulated at constant time, all algorithms have $\text{poly}(d/\epsilon)$ running-time (i.e., independent of the size of the graph). Actually, most algorithms have $\text{poly}(1/\epsilon)$ running-time and some have $\tilde{O}(1/\epsilon)$ running-time, where $\tilde{O}(\ell) = \text{poly}(\log(\ell)) \cdot \ell$. In particular, we present testing algorithms for the following properties:

connectivity: Our algorithm runs in time $\tilde{O}(1/\epsilon)$. Recall that by the above this means that in case the graph is connected the algorithm always accepts, whereas in case the graph is ϵ-far from being connected the algorithm rejects with probability at least $\frac{2}{3}$ and furthermore supplies a small counter-example to connectivity (in the form of an induced subgraph which is disconnected from the rest of the graph).

***k*-edge-connectivity:** Our algorithms run in time $\tilde{O}(k^3 \cdot \epsilon^{-3+\frac{2}{k}})$. For $k = 2, 3$ we have improved algorithms whose running-times are $\tilde{O}(\epsilon^{-1})$ and $\tilde{O}(\epsilon^{-2})$, respectively.

***k*-vertex-connectivity** (for $\boldsymbol{k = 2, 3}$)**:** Our algorithms run in time $\tilde{O}(\epsilon^{-k})$.

planarity: Our algorithm runs in time $\tilde{O}(d^4 \cdot \epsilon^{-1})$.

cycle-freeness: Our algorithm runs in time $\tilde{O}(\epsilon^{-3})$. Unlike all other algorithms, this algorithm has two-sided error probability, which is shown to be unavoidable for testing this property (within $o(\sqrt{N})$ queries, where N is the size of the graph).

In addition, we establish $\Omega(\sqrt{N})$ lower bounds on the query complexity of testing algorithms for the **Bipartite** and **Expander** properties. The first lower bound stands in sharp contrast to a result on testing bipartiteness which is described in [GGR96]. Recall that in [GGR96] graphs are represented by their $N \times N$ adjacency matrices, and the distance between two graphs is defined to be the fraction of entries on which their respective adjacency matrices differ. The Bipartite tester of [GGR96] works in time poly$(1/\epsilon)$ and distinguishes Bipartite graphs from graphs in which at least ϵN^2 edges must be omitted in order to be bipartite. Recall that in the current paper, graphs are represented by incidence lists of length d and distance is measured as the number of edge modifications divided by dN (rather than by N^2).

Finally, we observe that the known results on inapproximability of Minimum Vertex Cover (and Dominating Set) for bounded-degree graphs [ALM$^+$92, PY91], rule out the possibility of efficient testing algorithms for these properties in our model.

1.2 What does this type of approximation mean?

To make the discussion less abstract, let us consider the k-(edge)-connectivity tester. As evident from above, this algorithm is very fast; its running-time is polynomial in the error parameter, which one may think of as being a constant. Yet, what does one gain by using it?

One possible answer is that since the tester is so fast, it may make sense to run it before running an algorithm for k-connectivity. In case the graph is very far from being k-connected, we will obtain (w.h.p.) a proof towards this fact and save the time we might have used running the exact algorithm. (In case our tester detects no trace of non-k-connectivity, we may next run our exact algorithm.) It seems that in some natural setting where *typical* objects are either good or very bad, we may gain a lot. Furthermore, *if* it is *guaranteed* that objects are either good (i.e., graphs are k-connected) or very bad (i.e., far from being k-connected) then we may not even need the exact algorithm at all. The gain in such a setting is enormous.

Alternatively, we may be forced to take a decision, without having time to run an exact algorithm, while given the option of modifying the graph in the future, at a cost proportional to the number of added/omitted edges. For example, suppose you are given a graph which represents some design problem, where k-connectivity corresponds to a good design and changes in the design correspond to edge additions/omissions. Using a k-connectivity tester you always accept a good design, and reject with high probability designs which will cost a lot to modify. You may still accept bad designs, but then you know that it will not cost you much to modify them later. In this respect we mention the existence of efficient algorithms for determining a minimum set of edges to be added to a graph in order to make it k-connected [WN87, NGM90, Gab91, Ben95, NI96].

1.3 Testing connectivity to the rest of the graph

Our algorithm for testing k-edge-connectivity, for $k \geq 2$, uses a subroutine which may be of independent interest. To describe it, suppose that you are given as input a vertex which resides in a k-connected component of the graph separated from the rest of the graph by less than k edges. Your task is to find all vertices in the same component, and this should be done within complexity which only depends on the size of this component. As above, you are allowed oracle queries of the form "what is the i^{th} neighbor of vertex v".

Our algorithm finds the component containing the input vertex, within time cubic in the size of the component (independent of k and of the size of the entire graph). It is based on the underlying idea of the min-cut algorithm of Karger [Kar93]. For $k = 2$, we have an alternative algorithm which works in time linear in the size of the component.[3] We suggest the improvement of the complexity of the above task, for $k \geq 3$, as an open problem.

Organization

In Section 2 we present the definitions used throughout the paper. Section 3 presents our algorithms for testing k-edge-connectivity (for $k \geq 1$). Our algorithms for testing k-vertex-connectivity (for $k = 2, 3$) are presented in Section 4. Testing algorithms for Cycle-Free, Planar and Eulerian graphs are presented in Sections 5, 6 and 7, respectively. Our hardness results are presented in Section 8.

2 Definitions and Notation

We consider undirected graphs of bounded degree. We allow multiple edges but no self-loops. For a graph G, we denote by $\mathrm{V}(\mathrm{G})$ its vertex set and by $\mathrm{E}(\mathrm{G})$ its edge set. We assume, without loss of generality, that $\mathrm{V}(\mathrm{G}) = [|\mathrm{V}(\mathrm{G})|] \stackrel{\mathrm{def}}{=} \{1, \ldots, |\mathrm{V}(\mathrm{G})|\}$ and that for every vertex $v \in \mathrm{V}(\mathrm{G})$, there is an ordering among the edges incident to v. We stress that this ordering may be arbitrary and need not be consistent among neighboring vertices. Namely, $(u, v) \in \mathrm{E}(\mathrm{G})$ may be the i^{th} edge incident to u and the j^{th} edge incident to v, where $i \neq j$. In accordance with the above, we associate with a (bounded degree) graph G, a function $f_{\mathrm{G}} : \mathrm{V}(\mathrm{G}) \times [d] \mapsto \mathrm{V}(\mathrm{G}) \cup \{0\}$, where d is a bound on the degree of G. That is, $f_{\mathrm{G}}(v, i) = u$ if u is the i^{th} neighbor of vertex v and $f_{\mathrm{G}}(v, i) = 0$ if v has less than i neighbors.

We consider property testing algorithms which are allowed queries and work under the uniform distribution. Our measure of the (relative) distance between graphs depends on their degree bound. That is, the distance between two graphs G_1 and G_2 with degree bound d, where $\mathrm{V}(\mathrm{G}_1) = \mathrm{V}(\mathrm{G}_2) = [N]$, is defined as follows:

$$\mathrm{dist}_d(\mathrm{G}_1, \mathrm{G}_2) \stackrel{\mathrm{def}}{=} \frac{|\{(v, i) : \ v \in [N], \ i \in [d] \text{ and } f_{\mathrm{G}_1}(v, i) \neq f_{\mathrm{G}_2}(v, i)\}|}{d \cdot N} \tag{1}$$

This notation is extended naturally to a set, $\mathcal{C}$, of N-vertex graphs with degree bound d; that is, $\mathrm{dist}(\mathrm{G}, \mathcal{C}) \stackrel{\mathrm{def}}{=} \min_{\mathrm{G}' \in \mathcal{C}} \{\mathrm{dist}_d(\mathrm{G}, \mathrm{G}')\}$. For a graph property Π, we let $\Pi_{N,d}$ denote the class of graphs with N vertices and degree bound d which have property Π. In case $\Pi_{N,d}$ is empty for some Π, N, and d, we define $\mathrm{dist}(\mathrm{G}, \Pi_{N,d})$ to be 1 for every G.

Definition 2.1 *Let $\mathcal{A}$ be an algorithm which receives as input a size parameter $N \in \mathcal{N}$, a degree parameter $d \in \mathcal{N}$, and a distance parameter $0 < \epsilon \leq 1$. Fixing an arbitrary graph* G *with N vertices and degree bound d, the algorithm is also given oracle access to f_{G}. We say that $\mathcal{A}$ is a* property testing algorithm (or simply a testing algorithm) *for graph-property Π, if for every N, d, and ϵ and for every graph* G *with N vertices and maximum degree d, the following holds:*

- *if* G *has property Π then with probability at least $\frac{2}{3}$, algorithm $\mathcal{A}$* accepts G*;*
- *if* $\mathrm{dist}(\mathrm{G}, \Pi_{N,d}) > \epsilon$ *then with probability at least $\frac{2}{3}$, algorithm $\mathcal{A}$* rejects G.

In both cases, the probability is taken over the coin flips of $\mathcal{A}$.

[3] For $k = 3$, we present an algorithm which works in quadratic time.

In the above definition we deviate from some traditions of having also a confidence parameter, denoted δ, and requiring the testing algorithm to be correct with probability at least $1-\delta$.[4] of having also a confidence parameter, denoted δ, One can always obtain such a better performance at the cost of a multiplicative factor of $O(\log(1/\delta))$ in all complexities. We shall be interested in bounding both the query complexity and the running time of $\mathcal{A}$ as a function of N, d, and ϵ. In particular we try and achieve bounds which are polynomial in d, and $1/\epsilon$, and sub-linear in N. Actually, our query complexity will be independent of N and so is the running-time in a RAM model in which vertex names can be written, read and compared in constant time.

3 Testing k-Edge-Connectivity

Let $k \geq 1$ be an integer. A graph is said to be k–edge-connected if there are k edge-disjoint paths between each pair of vertices in the graph. An equivalent definition is that the subgraph resulting by omitting any $k-1$ edges (from the graph) is connected. A graph that is 1–edge-connected, is simply referred to as connected. In this section we show the following.

Theorem 3.1 *For every $k \geq 1$ there exists a testing algorithm for k-edge-connectivity whose query complexity and running time are* $\mathrm{poly}(\frac{k}{\epsilon})$. *Specifically,*

- *For $k = 1, 2$ these complexities are $O\left(\frac{\log^2(1/(\epsilon d))}{\epsilon}\right)$.*
- *For $k = 3$ these complexities are $O\left(\frac{\log^2(1/(\epsilon d))}{\epsilon^2 d}\right)$.*
- *For $k \geq 4$ these complexities are $O\left(\frac{k^3 \log^2(1/(\epsilon d))}{\epsilon^{3-\frac{2}{k}} d^{2-\frac{2}{k}}}\right)$.*

Furthermore, the algorithms never reject a k–edge-connected graph.

We note that the above complexity bounds do not increase with the degree bound d. The reason is that the distance between graphs is measured as a fraction of $d \cdot N$; thus, d effects the number of operations as well as the distance and its effect on the latter is typically more substantial.

We start by describing and analyzing the algorithm for $k = 1$, and later show how it can be generalized to larger k. From now on we assume that $d \geq k$, since otherwise we would immediately reject the tested graph G simply because a graph of degree less than k cannot be k connected. In the case of $k = 1$ we may actually assume that $d \geq 2$ (since otherwise, except for $N \leq 2$, the graph cannot be connected).

3.1 Testing Connectivity

Our algorithm is based on the following simple observation concerning the connected components (i.e., the maximal connected subgraphs) of a graph.

Lemma 3.1 *Let $d \geq 2$. If a graph G is ϵ-far from the class of connected graphs of degree bound d, then it has more that $\frac{\epsilon}{4} dN$ connected components.*

[4] Adopting these traditions seems justifiable in case one can derive better results than by mere repetition of the basic procedure. Alas, this is not the case in the present work.

The lemma is very easy to establish in case the maximum degree of G is below d. Otherwise, an additional argument is needed.

Proof: Assume contrary to the claim that G has at most $\frac{\epsilon}{4}dN$ connected components. We will show that by adding and removing at most $\frac{\epsilon}{2}dN$ edges we can transform G into a connected graph G′ which has maximum degree d. (Recall that according to our distance measure (Equation (1)) every edge in the symmetric difference between graphs is counted twice).

Let $C_1, \ldots, C_\ell$ be the connected components of G. The easy case is when the sum of degrees in each C_i is at most $d \cdot |C_i| - 2$. In this case, for every $i = 1, ..., \ell - 1$, we can add an edge between some vertex of C_i and some vertex of C_{i+1}. This maintains the degree bound and makes the graph connected. The number of edges added in such a case is $\ell - 1 \leq \frac{\epsilon}{4}dN - 1 < \frac{\epsilon}{2}dN$. But in general, the above condition may not hold and we need to do slightly more.

Suppose that for some connected component, C_i, the sum of degrees is greater than $d \cdot |C_i| - 2$ (and hence we cannot add edges between C_i and $C_{i\pm1}$ without violating the degree bound). Let T_i be an arbitrary spanning tree of C_i. Since T_i has at least 2 leaves, and by our assumption regarding C_i at least one of them has degree $d \geq 2$, that vertex has an incident edge in C_i which is not an edge in T_i. We can remove this edge from G without disconnecting C_i and get two vertices in C_i which have degree less than d. It follows that by removing at most one edge from each component and adding an edge between every C_i and C_{i+1}, we obtain a connected graph G′ respecting the degree bound d, where the symmetric difference between E(G) and E(G′) is bounded above by $2 \cdot \frac{\epsilon dN}{4}$. ■

As an immediate corollary we get

Corollary 3.2 *If a graph* G *is ϵ-far from the class of connected graphs then it has at least $\frac{\epsilon dN}{8}$ connected components each containing less than $\frac{8}{\epsilon d}$ vertices.*

By using the fact that each connected component contains at least one vertex we conclude that if G is ϵ-far from the class of connected graphs then the probability that a uniformly chosen vertex belongs to a connected component which contains at most $\frac{8}{\epsilon d}$ vertices, is at least $\frac{\epsilon d}{8}$. Therefore, if we uniformly choose $m = \frac{16}{\epsilon d}$ vertices, then the probability that no chosen vertex belongs to a component of size less than $\frac{8}{\epsilon d}$ is bounded above by $(1 - \frac{\epsilon d}{8})^m < \frac{1}{4}$. This gives rise to the following testing algorithm, where we assume that $N > \frac{8}{\epsilon d}$ since otherwise we could determine if the graph is connected by inspecting the whole graph[5].

Connectivity Testing Algorithm

1. Uniformly choose a set of $m = \frac{16}{\epsilon d}$ vertices;
2. For each vertex s chosen perform a Breadth First Search (BFS)[6] starting from s until $\lceil \frac{8}{\epsilon d} \rceil$ vertices have been reached or no more new vertices can be reached (a small connected component has been found);
3. If any of the above searches found a small connected component then output REJECT, otherwise output ACCEPT.

Since a connected graph consists of a single component, the algorithm never rejects a connected graph. The query complexity and running time of the algorithm are $m \cdot \frac{8}{\epsilon d} \cdot d = O(\frac{1}{\epsilon^2 d})$. We note that the choice to perform a BFS is quite arbitrary, and that any other linear-time searching method (e.g., DFS) will do. The complexity of the Connectivity Tester can be improved by applying

[5] In this uninteresting case the query complexity and running time are bounded by $O(Nd) = O(\frac{1}{\epsilon})$).

[6] The search is performed by making queries of the form (v, i).

Corollary 3.2 more carefully. That is, suppose that G has at least $L \stackrel{\rm def}{=} \frac{\epsilon dN}{4}$ connected components. Then, there exists an $i \leq \ell \stackrel{\rm def}{=} \log_2(8/\epsilon d)$ so that G has at least $\frac{L}{2^i \ell}$ connected components of size ranging between 2^{i-1} and $2^i - 1$. This suggests the following improved algorithm:

Connectivity Testing Algorithm (Improved Version)

1. For $i = 1$ to $\log(8/(\epsilon d))$ do:
   (a) Uniformly choose a set of $m_i = \frac{32 \cdot \log(8/(\epsilon d))}{2^i \epsilon d}$ vertices;
   (b) For each vertex s chosen, perform a BFS starting from s until 2^i vertices have been reached or no new vertices can be reached.

2. If any of the above searches found a small connected component then output REJECT, otherwise output ACCEPT.

Lemma 3.3 *If G is ϵ-far from the class of connected graphs then the improved testing algorithm will reject it with probability at least $\frac{3}{4}$. The query complexity and running time of the algorithm are $O(\frac{\log^2(1/(\epsilon d))}{\epsilon})$.*

Proof: Let B_i be the set of connected components in G which contain at most $2^i - 1$ vertices and at least 2^{i-1} vertices. Let $\ell \stackrel{\rm def}{=} \log_2(8/\epsilon d)$. By Corollary 3.2 we know that $\sum_{i=1}^{\ell} |B_i| \geq \frac{\epsilon dN}{8}$. Hence, there exists an $i \leq \ell$ so that $|B_i| \geq \frac{\epsilon dN}{8\ell}$. Thus, the number of vertices residing in components belonging to B_i is at least $2^{i-1} \cdot |B_i|$. It follows that the probability of choosing a vertex s in one of these components is at least

$$\frac{2^{i-1} \cdot |B_i|}{N} \geq \frac{\epsilon d \cdot 2^i}{16\ell} = \frac{2}{m_i}$$

Thus, with probability at least $\frac{3}{4}$, a vertex s belonging to a component in B_i is chosen in iteration i of Step (2), and the BFS starting from s will discover a small connected component leading to the rejection of G. The query complexity and running-time of the algorithm are bounded by $\sum_{i=1}^{\ell} m_i \cdot 2^i \cdot d = O(\frac{\log^2(1/(\epsilon d))}{\epsilon})$. ■

3.2 Testing k-Connectivity for $k > 1$

The structure of the testing algorithm for k-Connectivity where $k > 1$ is similar to the structure of the Connectivity Tester (i.e., case $k = 1$): We uniformly choose a set of vertices and for each of these vertices we test if it belongs to a small component of the graphs which has a certain property (i.e., is separated from the rest of the graph by a cut of size less than k). Similarly to the $k = 1$ case, we show that if a graph is ϵ-far from being k–connected then it has many such components. In addition, we present an efficient procedure for recognizing such a component given a vertex which resides in it.

A subset of vertices $X \subseteq V$ is said to be k–edge-connected if there are k edge-disjoint paths between each pair of vertices in X. We stress that, in case $k \geq 3$, these paths may go through vertices not in X and that any singleton is defined to be k–edge-connected. The k–edge-connected classes of a graph G are maximal subsets of V(G) which are k–edge-connected, and each vertex in V(G) resides in exactly one such class. In the remaining of this subsection, whenever we say k–connected we mean k–edge-connected, and a k-class is a k–connected class.

3.2.1 The Combinatorics

We start by assuming that the graphs we test for k-connectivity are $(k-1)$–connected. We later (in Sec. 3.2.6) remove this assumption. In Appendix A we describe in more detail the structure of $(k-1)$–connected graphs in terms of their k-classes. Here we only state the facts necessary for our algorithms. Let G be a $(k-1)$–connected graph. Then we can define an auxiliary graph $\mathrm{T_G}$ [DW95] (based on the *cactus* structure of [DKL76]), which is a tree, such that for every k-class in G there is a corresponding (unique) node in $\mathrm{T_G}$. The tree $\mathrm{T_G}$ might include additional auxiliary nodes, but they are not leaves and we shall not be interested in them here. If G is k–connected, then $\mathrm{T_G}$ consists of a single node, corresponding to the vertex set of G. Otherwise, $\mathrm{T_G}$ has at least two leaves. The leaves of $\mathrm{T_G}$ play a central role in our algorithm. Each leaf corresponds to a k-class C of G which is separated from the rest of the graph by a cut of size $k-1$. (Recall that G is assumed to be $(k-1)$–connected.) As we show below, for every leaf class C, given a vertex $v \in \mathrm{C}$, we can efficiently identify that v belongs to a leaf class. For $k=2$ this can be done deterministically within query and time complexity $O(|\mathrm{C}| \cdot d)$. For $k=3$ this can be done deterministically within query and time complexity $O(|\mathrm{C}|^2 \cdot d)$. For $k \geq 4$, we present a randomized algorithm with query and time complexity $O(|\mathrm{C}|^3 \cdot d)$. The analysis of our algorithm relies on the following lemma which directly follows from Lemma A.4 (see Appendix A).

Lemma 3.4 *Let* G *be a* $(k-1)$*–connected graph which is* ϵ*-far from the class of* k*–connected graphs with maximum degree* d*. Suppose that either* $d \geq k+1$ *or* $k \cdot |\mathrm{V(G)}|$ *is even.*[7] *Then,* $\mathrm{T_G}$ *has at least* $\frac{\epsilon}{8} dN$ *leaves.*

3.2.2 The Algorithm

Similarly to the $k=1$ case, the above lemma implies that at least $\frac{\epsilon dN}{16}$ of the leaves in $\mathrm{T_G}$ contain at most $\frac{16}{\epsilon d}$ vertices. Hence we can run the following algorithm, where the implementation of Step (2) is discussed subsequently. As was shown for the $k=1$ case, the algorithm below can be modified to save a factor of $\tilde{\Theta}(1/\epsilon d)$ in its query complexity and running time, but for sake of simplicity we describe the less efficient algorithm. We also assume that the number of vertices in G is greater than $\frac{32}{\epsilon d}$, since otherwise we could decide if the graph is k–connected by observing the whole graph and running an algorithm for finding a minimum cut (in time $\tilde{O}(Ndk)$ [Gab95]).

k-Connectivity Testing Algorithm

1. Uniformly choose a set of $m = \frac{32}{\epsilon d}$ vertices;
2. For each vertex s chosen, check whether s belongs to a leaf class which has at most $\frac{16}{\epsilon d}$ vertices.
3. If any leaf class was discovered then output REJECT, otherwise output ACCEPT.

As said above, this algorithm can be modified analogously to the improved version of the connectivity tester, yielding

[7] The reason for this technical requirement is to rule out the pathological case in which $d(=k)$ and $|\mathrm{V}(G)|$ are both odd in which case it is not possible to transform G into a k–connected graph with maximum degree d by performing edge modifications. In other words, the class of k–connected graphs with max-degree k where k and N are odd is empty. Clearly, this pathological case is easily detected by the algorithm.

Lemma 3.5 *The* (modified) *k-connectivity algorithm runs in time* $O(\frac{\log(1/(\epsilon d))}{\epsilon d}) \cdot \sum_{i=1}^{\log_2(16/(\epsilon d)} \frac{T_k(2^i)}{2^i}$, *where $T_k(n)$ is the time needed to implement the identification of a k-class leaf of size at most n* (i.e., Step (2)). *It always accept a k–connected graph and rejects with probability at least $\frac{2}{3}$ any graph which is $(k-1)$–connected but ϵ-far from being k–connected.*

In the following three subsection, we present such (k-class leaf) identification algorithms for the three cases $k = 2$, $k = 3$ and $k \geq 4$. The running-time bounds are $T_2(n) = O(nd)$, $T_3(n) = O(n^2 d)$, and $T_k(n) = O(n^{3-\frac{2}{k}} d)$, respectively, where d is the degree bound (or actually the maximum degree of vertices in the class).

3.2.3 Identifying a 2-class Leaf

Given a vertex s and an integer n, the following Identification Procedure can be used to determine whether s belongs to a 2–connected class of size at most n which is a leaf in $\mathrm{T_G}$. Note that the upper bound, n, on the size of the class is determined by the algorithm when calling the identification procedure.

2-Class Leaf Identification Procedure

1. Starting from s, perform a Depth First Search (DFS) until $n + 1$ vertices have been reached. Let $\mathrm{T_1}$ be the tree defined by the search, and let $\mathrm{E(T_1)}$ be its tree edges.
2. Starting once again from s, perform another search (using either DFS or BFS) until n vertices are reached or no new vertices can be reached. This search is restricted as follows: If (u, v) is an edge in $\mathrm{T_1}$, where u is the parent of v, then (u, v) cannot be used to get from u to v in the second search (but can be used to get from v to u). Let $\mathrm{X_2}$ be the set of vertices reached.
3. If there is a single edge with one end-point in $\mathrm{X_2}$ and the other outside of $\mathrm{X_2}$ (i.e. $(\mathrm{X_2}, \overline{\mathrm{X_2}})$[8] is a cut of size 1), then $\mathrm{X_2}$ is the 2-class s belongs to.

Clearly, the query complexity and running time of the procedure are $O(nd)$. Since the procedure always checks if it has found a cut of size 1, it will never identify a 2-class leaf when given a vertex s belonging to a 2-connected graph. Thus, we only need to prove the following.

Lemma 3.6 *Let* G *be a connected graph,* C *a* 2*-class in* G *of size at most n which is a leaf in* $\mathrm{T_G}$*, and s a vertex in* C*. Then the above procedure will always terminate with* $\mathrm{X_2} = \mathrm{C}$*.*

Proof: Since the first DFS terminates after seeing $n + 1$ vertices, and vertices in $\overline{\mathrm{C}}$ can be reached only by traversing the single edge (u, v) where $u \in \mathrm{C}$ and $v \in \overline{\mathrm{C}}$, we know that (u, v) must be a edge in T (with u being the parent). This ensures that the second search will never exit C. In other words, $\mathrm{X_2} \subseteq \mathrm{C}$. What needs to be shown is that the second search reaches *every* vertex in C (i.e., $\mathrm{X_2} = \mathrm{C}$), and hence the cut $(\mathrm{C}, \overline{\mathrm{C}})$ is discovered.

Assume contrary to this claim, that $\mathrm{S} \stackrel{\mathrm{def}}{=} \mathrm{C} \setminus \mathrm{X_2}$ is non-empty. Let $(u_1, v_1), \ldots, (u_\ell, v_\ell)$ be the set of edges crossing the cut $(\mathrm{X_2}, \mathrm{S})$, where $\forall i$, $u_i \in \mathrm{X_2}$ and $v_i \in \mathrm{S}$. Since C is 2–connected, there must be at least two edges in the cut $(\mathrm{X_2}, \mathrm{S})$. By our assumption that S is not reached in the second search, it follows that for every i, (u_i, v_i) is an edge in the DFS-tree T, and furthermore, u_i is the parent of v_i. However, since C is 2–connected there must be a path between v_1 and v_2 which

[8] For a subset $\mathrm{X} \subseteq \mathrm{V}$, we let $\overline{\mathrm{X}} \stackrel{\mathrm{def}}{=} \mathrm{V} \setminus \mathrm{X}$.

does not use the edge (u_1, v_1). There are two cases. In case the path does not contain vertices in X_2, we reach a contradiction to T being a DFS-tree. Otherwise, there must be a cut edge between some vertex, v, in the DFS-subtree rooted at v_1 and a vertex, u, in X_2. By the structure of the DFS-tree, this cannot be a DFS-tree edge from u to v, contradicting our hypothesis about the cut edges. ■

3.2.4 Identifying a 3-class Leaf

Given a vertex s and a size bound n, we first perform a DFS until $n+1$ vertices are discovered. At this point for each edge e in the tree (note that there are only n such edges) we "omit" e from the graph. (That is, in the rest of the algorithm we pretend that this edge is not in the graph.) Next we invoke the Identification procedure of the previous subsection (again starting from vertex s).

Lemma 3.7 *Let* G *be a* 2*–connected graph,* C *a 3-class leaf of* T_G *with at most n vertices, and s an arbitrary vertex in* C*. Then the above search process terminates in finding the cut* $(C, \overline{C})$*.*

It follows that we can identify a 3-Class Leaf of size n in time $O(n^2 d)$.

Proof: Clearly the initial DFS must cross an edge of the cut $(C, \overline{C})$, and so its DFS-tree has at least one cut edge. When this cut edge is omitted from the graph, the cut $(C, \overline{C})$ contains a single edge in the resulting graph, denoted G'. While the removal of this edge might decrease the connectivity of the vertices in C (which was 3 in G), they are at least 2–connected in G'. Invoking Lemma 3.6, we are done. ■

3.2.5 Identifying a k-class Leaf ($k \geq 2$)

The following applies to any $k \geq 2$, but for $k = 2, 3$ we have described more efficient procedures (above).

The algorithm for finding leaf k-classes ($k \geq 2$) is based on Karger's Contraction Algorithm [Kar93] which is a randomized algorithm for finding a minimum cut in a graph. Given a vertex s and a size bound n, the following search process is performed $\Theta(n^{2-\frac{2}{k}})$ times, or until a cut $(S, \overline{S})$ of size less than k is found: Starting from the singleton set $\{s\}$, at each step the algorithm has a set S of vertices it has visited. As long as $|S| < n$ and the cut $(S, \overline{S})$ has size at least k, the algorithm chooses an edge to traverse among the cut edges in $(S, \overline{S})$ and adds the new vertex reached to S. The cut edge chosen is the one having the smallest cost, where edges are assigned random costs as follows. Whenever a new vertex is added to S, its incident edges which were not yet assigned costs are each assigned a random cost uniformly in $[0, 1]$. Note that, as in the case of $k = 1$, the algorithm never rejects a k–connected graph (simply since a k–connected graph does not have any cut of size less than k).

Lemma 3.8 *Let* G *be a* $(k-1)$*–connected graph,* C *k-class leaf of* T_G *with at most n vertices, and s an arbitrary vertex in* C*. Then, with probability at least* $(2n)^{-(2-\frac{2}{k})}$*, a single iteration of the above search process succeeds in finding the cut* $(C, \overline{C})$*.*

Proof: Assume first that instead of assigning the edges costs in an online manner as described above, all edges in the graph are assigned random costs off-line. (We may think of our algorithm as simply revealing these costs as it proceeds.) Consider any assignment of costs to all edges in the graph. A spanning tree, T, of the subgraph induced by C is said to be **cheaper than the cut** if the cost of every edge in T is smaller than the cost of any of the cut edges between C and $\overline{\mathrm{C}}$.

Claim 3.8.1: Suppose that C contains a spanning tree which is cheaper than the cut $(\mathrm{C}, \overline{\mathrm{C}})$. Then the search process succeeds in finding $(\mathrm{C}, \overline{\mathrm{C}})$.

Comment: The above claim presents a sufficient but NOT necessary condition for the success of the search process. For example, the search may expand S by an edge with cost greater than any cut-edge in case S is not incident to any cut-edge.

Proof of Claim 3.8.1: By induction on the size of S. □

Thus, all we need is to lower bound the probability that C contains a cheaper-than-the-cut spanning tree. This is done by using Karger's analysis of his contraction algorithm (for finding a minimum cut) [Kar93]. Details follow.

We start by considering an auxiliary graph G′, in which all of $\overline{\mathrm{C}}$ is represented by an auxiliary vertex, denoted x. That is, $\mathrm{V}(G') = \mathrm{C} \cup \{x\}$ and $\mathrm{E}(G')$ contains all edges internal to C and an edge (u, x) for every edge (u, v) crossing the cut $(\mathrm{C}, \overline{\mathrm{C}})$ in G. Since C is a k–connected class in G, the graph G′ has a single minimum cut of size $k - 1$; that is, the cut $(\mathrm{C}, \{x\})$.

We now turn to Karger's analysis of his Contraction Algorithm. *Contraction* is an operation performed on a pair of vertices connected by an edge. When two vertices u and v are contracted, they are merged into a single vertex, w, where for each edge (u, z) such that $z \neq v$, we have an edge (w, z), and similarly for each edge (v, z') (such that $z' \neq u$). Thus multiple edges are allowed, but there are no self-loops. Given a graph as input, the Contraction Algorithm performs the following process until two vertices remain: It chooses an edge at random from the current graph (which is initially the original graph), and contracts its endpoints (resulting in a new graph which is smaller). An alternative presentation is to assign all edges uniformly chosen costs in $[0, 1]$ and to contract the cheapest edge at each step. Karger shows that the probability that the algorithm never contracts a min-cut edge is at least $2n^{-2}$. In our case, this means that with probability at least $2n^{-2}$, Karger's algorithm does not contract an edge incident to x, which implies that C has a spanning tree cheaper than the cut $(\mathrm{C}, \{x\})$.

To obtain the better bound claimed in the lemma, we reproduce Karger's analysis [Kar93]. He considers an n-vertex graph with min-cut of size c and such that the degree of every vertex in the residual graph at any step of the Contraction Algorithm is at least $D \geq c$. Hence, at the i^{th} step of the algorithm, the probability of choosing to contract a cut edge is at most $\frac{c}{(n-i)D/2}$. The probability no cut edge is contracted in any step of the algorithm is at least

$$\prod_{i=0}^{n-3} \left(1 - \frac{2c}{(n-i)D}\right) = \prod_{i=0}^{n-3} \left(\frac{n - i - (2c/D)}{n - i}\right) > (2n)^{-2c/D} \qquad (2)$$

where the strict inequality is due to elementary algebraic manipulations (see Appendix C). In our case, since all cuts in G′ other than the minimum cut $(\mathrm{C}, \{x\})$ have size at least k, we can set $c = k - 1$, $D = k$, and the lemma follows. ■

3.2.6 Testing k-Connectivity of Graphs which are not $(k - 1)$–connected

In the general case where the tested graph is not necessarily $k - 1$ connected, we claim that we can simply run the k-connectivity testing algorithm with distance parameter set to $\epsilon/O(k)$. Note that, for every $k \geq 4$ and $i \geq 1$, when we run the k-connectivity algorithm on an $(i-1)$–connected graph which is ϵ-far from being i–connected, the algorithm detects a cut of size $i-1$ with probability at least $\frac{2}{3}$. (We stress that this holds also for $i = 1$, in which case this means that the algorithm detects a small connected component.) Furthermore, the more efficient Identification procedures for 2-class and 3-class can be easily modified so that they remain valid when omitting edges. Specifically, in Step 1 of the 2-Class procedure, one should declare detection in case less than $n+1$ vertices are found in the initial DFS. The 3-Class procedure is modified analogously.

However, in general the situation may be more complex. The tested graph may not be $(i-1)$–connected for any $i \geq 1$ and we need to analyze what happens if we run the k-connectivity tester on such a graph. The following lemma allows us to simplify the analysis by considering the distance of the graph to the class of i–connected graphs rather than to the class of i–connected graphs with degree bound d.

Lemma 3.9 *Let* G *be a graph which is ϵ-far from the class of k-connected graphs with maximum degree d, where either kN is even or $d \geq k+1$.*[9] *Then the minumum number of edges which must be added to* G *in order to transform it into a k-connected graph* (without any bound on its degree), *is at least $\frac{1}{26}\epsilon dN$.*

Proof: Assume, contrary to the claim that in order to transform G into a k-connected graph it suffices to augment it with $m < \frac{1}{26}\epsilon dN$ edges. We next show that by adding and removing at most $13m$ edges we can transform G into a k-connected graph which has maximum degree d, in contradiction to the hypothesis.

Let G_k be a k-connected graph which results from augmenting G with m edges. Some of the vertices in G_k might have degree larger than d. Hence we define the *excess* of G_k (with respect to the degree bound d) as $\sum_{v,\ \deg(v)>d}(\deg(v)-d)$. Since G has maximum degree d, and G_k was obtained by augmenting G with m edges, the excess of G_k is at most $2m$. We now show how by performing at most $12m$ edge modifications to G_k, we can obtain a k-connected graph with excess 0 (i.e., maximum degree at most d). Thus, we transform G (via G_k) into a k-connected graph with degree bound d by modifying at most $m + 12m$ edges. At each step of the following process we decrease the excess of the graph while retaining its k-connectivity.

While the excess of the graph is non-zero, do:

1. If there is an edge (u, v) such that $\deg(u) > d$ and $\deg(v) > k$, remove (u, v). In case the graph remains k-connected, no additional modification is needed. Otherwise (the graph becomes $(k-1)$-connected), by Lemma A.2 (in Appendix A), the auxiliary tree of the graph consists of a simple path, with u belonging to one k-class leaf, and v to the other. Since v now has degree at least k, it cannot be a singleton leaf (because leaves have exactly $k-1$ edges going out of them). The same holds for u which now has degree at least $d \geq k$. We can thus apply Lemma A.3 on the two leaf k-classes, and obtain a k-connected graph at the cost of 4 edge modifications. Thus, we have decreased the excess by at least 1, at the cost of 5 edge modifications.

2. Otherwise, for every vertex u such that $\deg(u) > d$, all of u's neighbors have degree k (no vertex may have degree lower than k since the graph is k-connected). We consider two subcases.

[9] Recall that the technical condition (i.e., either kN is even or $d \geq k+1$) is required as otherwise the class of k-connected graph with maximum degree d is empty.

(a) If there are at least two such vertices u_1 and u_2 (i.e., with $\deg(u_i) > d$), then there must exist two vertices $v_1 \neq v_2$ such that v_1 is a neighbor of u_1 and v_2 is a neighbor of u_2. (If u_1 and u_2 only had a single (common) neighbor, or had edges between themselves, this would contradict the hypothesis that they both only have degree k neighbors.) We add an edge between v_1 and v_2, increasing their degree to $k+1$, and then apply Step 1 twice; that is, to the edges (u_i, v_i), for $i = 1, 2$. We have decreased the excess of the graph by 2, at a cost of $1 + 2 \cdot 5 = 11$ edge modifications.

(b) Otherwise, there exists a single vertex u with degree greater than d. Here we further consider two subcases.

  i. $\deg(u) > d + 1$. In such a case, we must remove at least two edges adjacent to u. Let $v_1 \neq v_2$ be any two neighbors of u (once again, the existence of two such distinct vertices follows from the hypothesis that all of u's neighbors have degree k). We now proceed as in Step 2.a, by adding an edge between v_1 and v_2 and then applying Step 1 to (u, v_1) and then to (u, v_2). We have decreased the excess of the graph by 2, at a cost of $1 + 2 \cdot 5 = 11$ edge modifications.

  ii. $\deg(u) = d + 1$. Let v be any neighbor of u (which, recall, must has degree k). In case there exists a vertex (other than v), denoted w, with degree smaller than d, we add an edge between v and w, raising the degree of v to $k+1$ (where the degree of w is now at most d). Applying Step 1 to the edge (u, v) we are done (at a cost of $1 + 5$ edge modifications).

  Otherwise, except for u and v, all vertices in the graph have degree d. We show that this is not possible by using the lemma's technical assumptions by which either $d > k$ or kN is even. In case $d > k$, all neighbors of u other than v have degree $d > k$, contradicting the hypothesis that all of u's neighbors have degree k (and again, u must have such neighbors since $\deg(v) = k < d + 1\deg(u)$). In case $d = k$ we have that u has degree $d + 1$ and all other vertices in the graph have degree $k = d$, yielding a degree sum of $kN + 1$ which is odd.

Thus in all cases, a decrease of 1 unit in the excess of the graph is obtained at a cost of at most 6 edge modifications. Since the initial excess is at most $2m$, the lemma follows. ■

Let G be ϵ-away from the class of k–connected graphs of degree bound d. By the above lemma, $m \geq \frac{\epsilon dN}{26}$ edges must be added to G to make it k–connected. For every $i \geq 1$, let us denote by m_i the minimum number of edges which should be added to G in order to make it i–connected, and let G_i denote an i–connected graph which results when adding such m_i edges to G. Let $m_0 \stackrel{\text{def}}{=} 0$ and $\mathrm{G}_0 \stackrel{\text{def}}{=} \mathrm{G}$. Then, there must exist an $i \in \{1, ..., k\}$ so that $m_i - m_{i-1} \geq m/k$. Let us consider any such i and let $\epsilon' \stackrel{\text{def}}{=} \epsilon/(26k)$. It follows that in order to transform G_{i-1} into an i–connected graph, we must augment it with at least $\epsilon' dN$ edges. This implies that the auxiliary tree of G_{i-1} has a least $\frac{1}{2}\epsilon' dN$ leaves, and so, had we run the k-connectivity tester on G_{i-1} with approximation parameter ϵ', it would detect that G_{i-1} is not k ($> i$) connected, with probability at least $\frac{2}{3}$. What is left to show is that the detection probability of the k-connectivity tester on the graph G, which is a subgraph of G_{i-1}, is no smaller. Although this sounds very appealing, a proof is in place. Actually we will modify the analysis of the detection probability of G_{i-1} so that it applies to G.

Recall that our analysis of the execution of the algorithm on an $(i-1)$–connected graph only refers to the number of leaf i-classes of certain small sizes. Specifically, a leaf i-class C is hit with probability $\frac{|\mathrm{C}|}{N}$ and is identified as such (with high probability) within time $T_i(|\mathrm{C}|)$ (see Sec. 3.2.2). Note that C is not necessarily a (leaf) i-class in G (as the structure of i-classes in G may be very different than in G_{i-1} and in particular G may not be $(i-1)$–connected). Instead we let C' be a minimal subset of C which is separated from the rest of G by a minimal number of edges, denoted j. Such a set is sometimes referred to as *j-extreme*. Since in G_{i-1} the whole set C is separated from the rest of the graph by $i-1$ edges, we have that $j \leq i-1$. Furthermore, by the definition of C', it contains no (strict) subset which is separated from the rest of G by less than j edges. Thus, we may apply the analysis of Sec. 3.2.5 to C'. It follows that if a vertex $s \in C'$ is chosen by the (modified) algorithm in iteration $\ell = \lceil \log(|C'|) \rceil$ (i.e. when testing if the graph has many leaves of size at most $2^\ell - 1$ and at least $2^{\ell-1}$), then the leaf identification procedure, starting from s, detects the cut $(C', \overline{C'})$, with high probability, within time $T_j(2 \cdot |C'|)$. The above analysis holds also with respect to the (modified) Identification procedures for 2-class and 3-class.

4 Testing k-Vertex-Connectivity for $k = 2, 3$

The definitions for vertex-connectivity are analogous to the ones for edge-connectivity. There are also similarities in the induced structures, though the structures induced by vertex-connected classes tend to be more complex. In particular, although we believe that our techniques will apply to arbitrary k, we have only verified the relatively simpler cases of $k = 2, 3$.

For $k \geq 1$, a graph G having at least $k+1$ vertices is said to be **k-vertex-connected** if there are k vertex-disjoint paths between each pair of vertices in G. An equivalent definition is that the subgraph of G resulting by omitting any $k-1$ vertices (and the edges incident to them) is connected. For $k = 1$, edge-connectivity and vertex-connectivity coincide, but for $k \geq 2$ the two notions are quite different. Assume from now on that $|V(G)| \geq k+1$.

Theorem 4.1 *For $k = 2, 3$ there exists a testing algorithm for k-vertex-connectivity whose query complexity and running time are* $\mathrm{poly}(1/\epsilon)$. *In particular,*

1. *For $k = 2$ these complexities are*

$$\min\left\{ O\left(\frac{\log(1/(\epsilon d))}{\epsilon^2 d}\right), O\left(\frac{2^d \log^2(1/(\epsilon d))}{\epsilon}\right) \right\}$$

2. *For $k = 3$ these complexities are*

$$\min\left\{ O\left(\frac{\log(1/(\epsilon d))}{\epsilon^3 d^2}\right), O\left(\frac{2^{2d} \log(1/(\epsilon d))}{\epsilon^2 d}\right) \right\}$$

Similarly to the case of edge-connectivity, our vertex connectivity testing algorithms try to find small k-vertex-connected classes. A subset of vertices $X \subseteq V$ is said to be **k-vertex-connected** (**k-connected**) if there are k vertex-disjoint paths between each pair of vertices in X. As is the case for k-edge-connectivity, when $k \geq 3$, these paths may pass through vertices not in X. The **k-vertex-connected classes** (**k-classes**) of a graph G are maximal subsets of V(G) which are k-vertex-connected. In contrast to edge-connected classes, a vertex may belong to several vertex-connected classes. However, every two k-vertex-connected classes of a graph can have at most $k-1$ common vertices.

For $k = 2, 3$, given a $(k-1)$-connected graph G, we can define an auxiliary graph T_G which is a tree. Similarly to the case of edge-connectivity, the leaves of the tree will play an important role in our algorithms. In particular we'll be interested in identifying leaves of T_G which correspond to k–classes of G which contain at least $k+1$ vertices, and leaves which correspond to sets which contain a vertex with only $k-1$ distinct neighbors. The former, which we'll refer to as k–class leaves, have the property that they contain a single separating set of size $k-1$ – i.e., a set of $k-1$ vertices whose removal disconnects the graph. For more details on the structure of the auxiliary tree of 2 and 3 connected graphs, see [Eve79] and [Pou92], respectively. For our purposes we only need the above stated fact concerning the k-class leaves and the following lemma which follows from Lemmas B.4 and B.8 (see Appendix B).

Lemma 4.1 1. *Let* G *be a connected graph which is ϵ-far from the class of* 2*-connected graphs. Then the sum of the number of degree-1 vertices in* G *and the number of 2-class leaves in* $\mathrm{T_G}$ *is at least* $\frac{\epsilon dN}{6}$.

2. *Let* G *be a* 2*-connected graph which is ϵ-far from the class of* 3*-connected graphs with maximum degree d. If $d \geq 4$ or* G *has an even number of vertices,*[10] *then the sum of the number of degree-2 vertices in* G *and the number of 3-class leaves in* $\mathrm{T_G}$ *is at least* $\frac{\epsilon dN}{8}$.

The vertex-connectivity testing algorithms have the same structure as the edge-connectivity testing algorithms. Namely, for both $k=2$ and $k=3$ we uniformly choose a set of $O(\frac{1}{\epsilon d})$ vertices, and for each vertex s chosen we first check if s has only $k-1$ different neighbors, in which case we immediately reject the graph. Otherwise (s has at least k neighbors), we run a procedure for checking if s belongs to a k-class leaf of size $O(1/(\epsilon d))$. Thus, a straightforward implementation would run in time $O\left(\frac{1}{\epsilon d} \cdot T_k(O(\frac{1}{\epsilon d}))\right)$, where $T_k(\cdot)$ is the running time of the identification procedure. Using the same technique described in the edge connectivity testing algorithms, we can cut a factor of $\tilde{\Theta}(\frac{1}{\epsilon d})$ in the running time. We hence focus on describing how to identify a small k–class leaf given a vertex in the class.

4.1 Identifying a 2-class Leaf

We have two procedures for identifying a 2-class leaf C given a vertex $s \in \mathrm{C}$ and an upper bound n of the size of C. The first has running time $O(n^2 \cdot d)$, and the second has running time $O(n \cdot d \cdot 2^d)$.

2-Class Leaf Identification Procedure (Version I)

1. Perform a BFS (or DFS) starting from s until n vertices are reached. Let the set of vertices reached be denoted by X.
2. For each vertex $v \in \mathrm{X} \setminus \{s\}$, start a new search from s in the auxiliary graph resulting from the omission of vertex v from the given graph. That is, start a new BFS from s, except that when vertex v is reached treat it as if it has no other incident edges (i.e., do not extend the search from it). The search is terminated once $n+1$ vertices (including s and v) were reached. Let us denote by X_v the set of vertices reached in this search (including v).
3. If for some $v \in \mathrm{X}$, the number of vertices in X_v is at most n, then X_v is a 2-class leaf.

Clearly, the query and time complexity of the above procedure are $O(n^2 d)$.

Lemma 4.2 *Let* G *be a connected graph with more than n vertices,* C *a* 2*-class in* G *of size at most n which is a leaf in* $\mathrm{T_G}$*, and s a vertex in* C *which is not a separating vertex. Then for some v chosen in Step (2) of the above procedure,* $\mathrm{X}_v = \mathrm{C}$. *On the other hand, if* G *is* 2*-connected and* $|\mathrm{V(G)}| > n$*, then the above procedure will never find a leaf class of size* $\leq n$.

[10] See Footnote 7.

Differently from the leaf identification procedures in the edge-connectivity case, here, in order to detect a leaf, the procedure cannot start from ANY vertex belonging to the leaf class. In particular, it should not start from a separating vertex. However, since each leaf class contains at least one non-separating vertex and exactly one separating vertex, the probability of choosing a good starting point is at least $\frac{1}{2}$ the probability of choosing any vertex in a leaf class. Hence our analysis is essentially unchanged.

Proof: Suppose first that G is 2-connected (and $|\mathrm{V}(\mathrm{G})| > n$). Then for every choice of v (in Step 2) of the procedure, there are connected paths not passing through v between vertex s and any other vertex in $\mathrm{V}(\mathrm{G}) \setminus \{s, v\}$. Thus, $|\mathrm{X}_v| > n$ for every v.

Consider now a non-separating vertex s which resides in a 2-class, C, of size at most n. Consider the first BFS performed in Step (1) of the procedure. Since it reaches n vertices, one of these vertices must be the single separating vertex belonging to C, which we denote by w. Since w is the only separating vertex in C, any vertex outside of C can be reached only by passing through w. This implies that when $v = w$ in Step (2) of the procedure, the set of vertices X_v is a subset of C. It remains to show that every vertex in C is reached in this execution of Step (2). But this follows directly from the fact that C is a two-connected class. ■

2-Class Leaf Identification Procedure (Version II) The second procedure for identifying a 2-class leaf is based on the the observation that the problem of identifying a 2-vertex-connected class leaf can be reduced to the problem of identifying a 2-edge-connected class leaf (or simply a cut of size 1) in an auxiliary graph. In particular, consider the following randomized transformation of the tested graph, G, into a new graph G′: Replace each vertex v in G by two vertices, v_1 and v_2, connected by an edge, and partition the edges incident to v randomly among v_1 and v_2. That is, a random non-trivial subset of these edges are now incident to v_1 and the rest are incident to v_2.

Consider a separating vertex, w, belonging to a leaf class C in G, and let C′ be the corresponding set of vertices in G′. That is, $\mathrm{C}' = \{v_i : v \in \mathrm{C},\ i \in \{1, 2\}\}$. Since C is 2-vertex-connected, w must have some (actually – at least two) incident edges (in G) whose other end-points are in C. Let this set of incident edges be denoted $\mathrm{E}_1(w)$, and the remaining edges (with end-points outside of C), be $\mathrm{E}_2(w)$. Suppose that when replacing w with two vertices, w_1 and w_2, the set of edges incident to w_1 is $\mathrm{E}_1(w)$ (and the set incident to w_2 is $\mathrm{E}_2(w)$). This event happens with probability at least 2^{-d}. Since w is a separating vertex, the edge between w_1 and w_2 is a *bridge* in G′ – that is, its removal disconnects the vertices in $\mathrm{C}'' \stackrel{\text{def}}{=} \mathrm{C}' \setminus \{w_2\}$ from the rest of G′. On the other hand, it is not hard to verify, that no matter how the other, non-separating, vertices in C are "broken into two", no other edge incident to a vertex in C″ is a bridge in G′.

Thus, suppose we have chosen a non-separating vertex s that belongs to a small leaf class C of G. We can now simulate the procedure for identifying a 2-edge-connected class in a graph G′, where we transform G into G′ randomly as we execute the procedure. Namely, whenever we encounter a new vertex v, we do the following. We rename v as v_1, and "virtually" connect it to another (new) vertex v_2. We then randomly partition v's set of incident edges into two non-trivial subsets, $\mathrm{E}_1(v)$ and $\mathrm{E}_2(v)$, so that the edge traversed in order to reach v belongs to $\mathrm{E}_1(v)$. We think of v_1 as being incident to $\mathrm{E}_1(v)$, and of v_2 as incident to $\mathrm{E}_2(v)$. The identification procedure (for a 2-edge-connected class leaf) treats virtual edges as real edges. In order to achieve success probability 2/3, for each vertex chosen by the algorithm we perform the above randomized process 2^{d+1} times.

4.2 Identifying a 3-class Leaf of a 2-Connected Graph

Analogously to the case of 2-class leaf identification, we use two alternative procedures for identifying 3-class leaves. Both procedures are straightforward extensions of the ideas presented in the 2-class case. Specifically, the first identification procedure performs three "levels" of BFS rather than two:

1. Perform a BFS starting from s until n vertices are reached. Let the set of vertices reached be denoted by X.
2. For every vertex $v \in \mathrm{X} \setminus \{s\}$:
   (a) Perform a BFS in the auxiliary graph resulting from the omission of vertex v. Again, the search is suspended once $n+1$ vertices are discovered. Denote the set of vertices reached (including v) by X_v.
   (b) For every vertex $w \in \mathrm{X}_v \setminus \{s, v\}$, perform a BFS in the auxiliary graph resulting from the omission of both v and w. Again, the search is suspended once $n+1$ vertices are discovered. Denote the set of vertices reached (including v and w) by $\mathrm{X}_{v,w}$.
3. If for some $v \in \mathrm{X}$ and $w \in \mathrm{X}_v$, the number of vertices in $\mathrm{X}_{v,w}$ is at most n, then $\mathrm{X}_{v,w}$ is a 2-class leaf.

The second identification procedure is based on the same reduction of the identification of vertex-connected-classes to the identification of edge-connected-classes. This time we consider two sets of edges – those incident to each of the two vertices which separate the class from the rest of the graph. With probability at least 2^{-2d} both sets are partitioned so that the edges going into the vertex-class are on one side and the rest of the edges (going to the rest of the graph) are on the other.

4.2.1 Testing k-Connectivity of Graphs which are not $(k-1)$-connected

Consider first the case in which $k = 2$ and the graph is not necessarily connected. We claim that in this case we may simply run the 2-vertex-connectivity testing algorithm with distance parameter set to $\frac{\epsilon}{8}$. Note that if the graph G is not connected, and the 2-class leaf identification procedure is given a vertex s belonging to a small connected component of G, then it will always output that it has identified a leaf[11]. Thus, in case G is $\frac{\epsilon}{8}$-far from being connected, with high probability, the testing algorithm will reject G.

Otherwise, let G_1 be a connected graph with maximum degree d which is at distance smaller than $\frac{\epsilon}{8}$ from G. Assuming that G is ϵ-far from the class of 2-connected graphs with maximum degree d, G_1 must be at least $\frac{7}{8}\epsilon$-far from this class. Thus, by Lemma 4.1, its auxiliary tree, $\mathrm{T}_{\mathrm{G}_1}$, has at least $\frac{7}{48}\epsilon dN$ 2-class leaves and vertices with a single neighbor. Let G'_1 be the graph whose edge set is the union of the edge sets of G and G_1. Clearly, G'_1 is connected (though its maximum degree might be larger than d). Furthermore, the edge set of G'_1 is a superset of both the edge set of G and the edge set of G_1, and it is at most $\frac{\epsilon}{16}dN$ larger than each one of them. In particular, this implies that $T_{\mathrm{G}'_1}$ has at least $\frac{7}{48}\epsilon dN - \frac{1}{8}\epsilon dN = \frac{1}{48}\epsilon dN$ 2-class leaves and vertices with a single neighbor (since the addition of an edge can remove at most two leaves from the tree). This means

[11] For sake of elegance, the procedure can explicitly check if in the first BFS it has reached less than n vertices, in which case it will stop and output that it has found a small connected component

that if we tested G'_1 for 2-connectivity, it would be rejected with high probability. Namely, with high probability, the algorithm would choose a vertex s such that either s has only one neighbor or s belongs to a small 2-class leaf C of G'_1 (but is not a separating vertex), and this leaf would be identified.

Now consider the actual algorithm which runs on G. Since the number of neighbors each vertex in G has is bounded by the number of neighbors it has in G'_1, the algorithm will reject the graph if a vertex s which has only one neighbor in G'_1 (and hence in G) is chosen. Thus consider a vertex s which belongs to a small 2-class leaf C in $T_{G'_1}$ (but is not a separating vertex), and assume the leaf identification procedure on G starts from s. Since the edge set of G is a subset of the edge set of G'_1, either s cannot reach the separating vertex v of C, in which case it belongs to a small connected component, or v is still a separating vertex in G. In both cases, the leaf identification procedure will detect it,

Similarly, for $k = 3$, it suffices to run the 3-connectivity testing algorithm with distance parameter set to $\frac{\epsilon}{80}$. By the discussion above concerning 2-connectivity, if the graph is at least $\frac{\epsilon}{10}$-far from being 2-connected, then it will be rejected with high probability. Otherwise, we use the same argument as above to show that there exists a 2-connected graph G'_2 whose edge set is a superset of the edge set of G, such that there are many 3-class leaves in $T_{G'_2}$. Similarly to the 2-class case, if the algorithm (executed on G) chooses a vertex s in one of these leaf classes C of G'_2, then it will either find that s belongs to a small connected component, or that there exist one or two vertices separating s (together with all or part of C) from the rest of the graph. It follows that G is rejected with high probability.

5 Testing if a Graph is Cycle-Free (a Forest)

The testing algorithm described in this section is based on the following observation. Let G be the tested graph and $C_1, C_2, \ldots, C_k$ its connected components. By definition, if G is cycle-free then each of its components is a tree. We should therefore expect each C_i to have $|C_i| - 1$ edges, and so the total number of edges in G should be $N - k$. Intuitively, if G is far from being cycle-free, then this is due mainly to either many extra edges within small components or to many extra edges inside big components. In the first case, we can hope to sample a bad small component. In the second case, we may consider the subgraph if G which consists of all big component and detect a discrepancy between its edge count and its vertex count. (Since here the number of components is relatively small it cannot account for this discrepancy.) Details follow.

Let C_i be the i^{th} connected component of G, and denote by m_i the number of edges in C_i. Denote $n_i \stackrel{\text{def}}{=} |C_i|$, and let $b_i \stackrel{\text{def}}{=} m_i - (n_i - 1) \geq 0$ be the number of edges which should be removed from C_i to make it a tree. Suppose that the components are arranged according to decreasing size and let t be the number of components of size at least $\frac{8}{\epsilon d}$ (i.e., $n_i \geq 8/\epsilon d$ iff $i \leq t$). Let $b \stackrel{\text{def}}{=} \sum_{i=1}^{k} b_i$ and consider the following two cases.

Case 1: Suppose $\sum_{i=1}^{t} b_i \leq b/2$. In this case we may forget of the big components and concentrate on finding a violation (cycle) inside a small component. If we select a vertex at random then it will belong to a small component with probability at least $\frac{b/2}{dN}$. Once we have selected such a vertex, we may detect a cycle in its component by conducting a search on the component. The complexity of the search is bounded by the size of the component; that is, the complexity is $\frac{8}{\epsilon d} \cdot d$.

Case 2: Suppose $\sum_{i=1}^t b_i > b/2$. In this case we may forget of the small components and concentrate on approximating the sum $\sum_{i=1}^t b_i$. This can be done by sampling vertices, and checking if they are inside a large component. This sampling enables us to estimate $\sum_{i=1}^t n_i$ (i.e., by the probability we fall inside a large component) as well as $\sum_{i=1}^t m_i$ (i.e., by the average of the degrees of vertices selected inside large components). A discrepancy of substantially more than t between the estimates (for $\sum_{i=1}^t n_i$ and $\sum_{i=1}^t m_i$) indicates a big distance from cycle-freeness.

Putting everything together, we get the following algorithm.

Cycle-Freeness Testing Algorithm

1. Uniformly choose a set of $\ell = \Theta(\frac{1}{\epsilon^2})$ vertices;
2. For each vertex s chosen, perform a BFS starting from s until $\frac{8}{\epsilon d}$ vertices are reached or no more new vertices can be reached (s belongs to a small connected component);
3. If any of the above searches found a cycle then output REJECT (otherwise continue);
4. Let $\hat{n}$ be the number of vertices in the sample which belong to connected components of size greater than $\frac{8}{\epsilon d}$, and let $\hat{m}$ be half the sum of their degrees. If $\frac{\hat{m}-\hat{n}}{\ell} \geq \frac{\epsilon d}{16}$ then output REJECT, otherwise output ACCEPT.

This establishes that:

Theorem 5.1 *There exists a testing algorithm for the Cycle-Free property whose query complexity and running time are* $O(\frac{1}{\epsilon^3 d})$.

Proof: Let us denote by t the number of *big* connected components (i.e., connected components of size at least $8/\epsilon d$). Firstly, note that with probability at least $\frac{2}{3}$ both estimates done in Step 4 are accurate to within $(\epsilon d)/32$; that is, $\frac{\hat{m}}{\ell} = \frac{M'}{N} \pm \frac{\epsilon d}{32}$ and $\frac{\hat{n}}{\ell} = \frac{N'}{N} \pm \frac{\epsilon d}{32}$, where N' (resp., M') is the number of vertices (resp., edges) in big components. From this point on we assume that these estimates are good.

In case G is cycle-free, we never reject in Step 2. Furthermore, in this case we have $M' - N' = -t \leq 0$ and so Step 4 makes us accept. On the other hand, if G is ϵ-far from cycle-free then either there are $\frac{\epsilon dN}{4}$ superfluous edges inside small components or there are $\frac{\epsilon dN}{4}$ superfluous edges inside large components. The first case is detected in Step 2 with probability at least $(1-\frac{\epsilon}{4})^\ell > \frac{2}{3}$, whereas the second case is detected by Step 4 provided the estimates are good. Specifically, in the latter case $M' - N' \geq \frac{\epsilon dN}{4} - t \geq \frac{\epsilon dN}{8}$. ■

REMARK: The above tester has two-sided error probability. This is unavoidable if one allows only $o(\sqrt{N})$ many queries. To see why consider either classes considered in the proof of Theorem 8.1: A $o(\sqrt{N})$-query algorithm must reject a random graph in the class with high probability and without seeing a cycle in it! Fixing any such sequence of coins, we observe that the algorithm will also reject a graph which consists only of the (partial) forest it has observed. Thus the algorithm has a non-zero rejecting probability on some cycle-free graphs. It is even easier to show that any $o(N)$-query algorithm must have a non-zero accepting probability on graphs which are far from cycle-free (e.g., consider the execution on the empty graph).

6 Testing Planarity

A graph is *planar* if it can be drawn in the plane so that no two edges in the graph cross each other (*cf.* [Eve79]). Our planarity testing algorithm is based on a theorem which is due to Kuratowski [Kur30]. Two graphs are said to be *homomorphic* if both can be obtained from the same graph by replacing edges with paths of degree-2 vertices (where these degree-2 vertices do not appear in the original graph). The graph $K_{3,3}$ is a completely connected bipartite graph with 3 vertices on each side, and the graph K_5 is a clique of 5 vertices.

Kuratowski's Theorem: *A graph* G *is planar if and only if no subgraph of* G *is homomorphic to either* $K_{3,3}$ *or* K_5*.*

We begin by considering the easier problem of testing whether a graph is H-free, where H is any fixed constant size graph (e.g., $K_{3,3}$ or K_5). A graph G is H-free, if no subgraph is G is *isomorphic* to H. Let diam(H) denote the diameter of H.

H-freeness Testing Algorithm

1. Choose uniformly a set of $m = \Theta(\frac{1}{\epsilon})$ vertices;
2. For each vertex s chosen, perform a BFS starting from s to depth diam(H).
3. If any of the above searches found a subgraph isomorphic to H then output REJECT, otherwise output ACCEPT.

Lemma 6.1 *The above algorithm is a testing algorithm for the* H*-freeness property whose query complexity and running time are* $O(\frac{d^{\mathrm{diam(H)}}}{\epsilon})$ *and* $O(\frac{d^{\mathrm{diam(H)}\cdot|\mathrm{H}|}}{\epsilon})$*, respectively.*

Proof: Clearly, if G is H-free it will be accepted with probability 1. Since in each search at most $d^{\mathrm{diam(H)}}$ queries are asked, the algorithm's query complexity is $O(\frac{d^{\mathrm{diam(H)}}}{\epsilon})$. The third step of the algorithm (looking for a subgraph isomorphic to H) can be performed in time $O(\frac{d^{\mathrm{diam(H)}\cdot|\mathrm{V}(H)|}}{\epsilon})$, by checking all mappings from the (at most) $d^{\mathrm{diam(H)}}$ vertices reached to H.

It thus remains to show that if G is ϵ-far from the class of H-free graphs then the H-freeness Testing Algorithm will reject it with probability at least $\frac{2}{3}$. But this follows directly from the definition of ϵ-far: If G is ϵ-far from the class of H-free graphs then it contains at least $\frac{\epsilon}{2}dN$ edges which each reside in at least one subgraph of G which is isomorphic to H. Since the degree of every vertex is at most d, there are at least $\frac{\epsilon}{2}N$ vertices which reside in such subgraphs. Since the algorithm uniformly chooses $\Theta(\frac{1}{\epsilon})$ vertices, with probability 2/3 at least one of these vertices resides in such a subgraph, and this will be detected in the third step of the algorithm. ■

If a graph is ϵ-far from the class of planar graphs, then it contains at least $\frac{\epsilon}{2}dN$ edges which reside in a subgraph of G which is homomorphic to either $K_{3,3}$ or K_5. Note that since neither $K_{3,3}$ nor K_5 have degree 2 vertices this means that such a subgraph can be obtained by replacing edges of $K_{3,3}$ or K_5 with paths of degree 2 vertices. Consider a particular edge (u, v) in a subgraph homomorphic to $K_{3,3}$ or K_5, and the corresponding homomorphism. Without loss of generality, let this homomorphism be to $K_{3,3}$. In this homomorphism, either (u, v) alone is mapped to an edge in $K_{3,3}$ (in which case both u and v have degree at least 3) or (u, v) belongs to a path which is mapped to an edge in $K_{3,3}$ (in which case either u or v has degree 2 (or possibly both have degree 2)). We thus need to replace the BFS in the H-freeness testing algorithm with a slightly different search. Suppose we could transform G into a *contracted* graph which contains no vertices with degree 2. Namely, every path in the graph in which all vertices except its endpoints have degree 2, is *contracted* into a single edge between the two endpoints. Note that all such paths are disjoint, and hence the process is well defined. We would thus like to essentially simulates a BFS to depth $\mathrm{diam}(K_{3,3}) = 2$ on the contracted graph, given access to G.

The only problem that arises if we actually perform this simulation is that some paths might be very long, causing the simulation to be expensive. Fortunately, there can't be too many long paths. More precisely, since every vertex is an intermediate vertex in at most one such path, there are no more than $\frac{\epsilon}{4}dN$ paths with more than $\frac{4}{\epsilon d}$ intermediate vertices (or edges). It follows that if G is ϵ-far from the class of planar graphs, then it contains at least $\frac{\epsilon}{4}dN$ edges that reside in a subgraph of G which is homomorphic to either $K_{3,3}$ or K_5 and do not belong to a path of length greater than $\frac{4}{\epsilon d}$. The above discussion gives rise to the following algorithm.

Planarity Testing Algorithm

1. Uniformly choose a set of $m = \Theta(\frac{1}{\epsilon})$ vertices;
2. For each vertex u chosen perform the following procedure.
   (a) If u has degree at least 3 then let $s = u$. If u has degree 1 then Stop (go to 2). Otherwise (u has degree 2), perform a DFS starting from u until a vertex with degree at least 3 is reached or $\frac{4}{\epsilon d} + 2$ vertices are reached (or no new vertices can be reached). If a degree 3 vertex is reached then let s be this vertex. Otherwise Stop (go to 2).
   (b) Starting from s perform a "BFS" as follows. Every vertex v reached, is assigned a label $\ell(v) = (\ell_1(v), \ell_2(v))$, where $\ell(s) = (0, 0)$ and the label assignment rule is defined as follows. If $v_1, \ldots, v_k$ are the children of v in the BFS tree, then: (1) For every v_i with degree 2, let $\ell_1(v_i) = \ell_1(v)$ and let $\ell_2(v_i) = \ell_2(v) + 1$; (2) For every v_i with degree other than 2, let $\ell_1(v_i) = \ell_1(v) + 1$, and $\ell_2(v_i) = 0$. The search should be discontinued at vertices v for which $\ell_1(v) = 2$, or $\ell_2(v) = \frac{4}{\epsilon d} + 1$.
3. If any of the above searches found a subgraph homomorphic to either $K_{3,3}$ or K_5 then output REJECT, otherwise output ACCEPT.

The correctness of the algorithm follows from Lemma 6.1 and the discussion following it. The number of queries performed is $O(\frac{1}{\epsilon})$ larger than that stated in Lemma 6.1 (where here diam(H) is 1 for K_5 and 2 for $K_{3,3}$), since we might need to follow paths of that length in our search (Step (2b)). As for the running time, we can obtained better bounds than those implied by Lemma 6.1 as follows. First note that for each starting vertex s, the graph induced by the search in Step (2b) can be contracted while performing the search. Thus Step (3) is reduced to determining whether some contracted subgraph containing s is isomorphic to $K_{3,3}$ or K_5. For the former we have to check if there exist three neighbors of s which all have two common neighbors (other than s). For the latter we can simply go over all subsets $\{v_1, v_2, v_3, v_4\}$ of 4 neighbors of s and check if $\{s, v_1, v_2, v_3, v_4\}$ induces a clique.

As described above, the algorithm has query complexity $O(\frac{d^2}{\epsilon^2})$ and running time $O(\frac{d^2}{\epsilon^2} + \frac{d^4}{\epsilon})$. However, similarly to the connectivity algorithms described in Sections 3 and 4, we can save a factor of $\tilde{\Theta}(1/\epsilon)$ in the query complexity (and in the first term of the bound on the running time).

Theorem 6.1 *There exists an algorithm for testing planarity whose query complexity and running are $O(\frac{d^2\log^2(1/\epsilon)}{\epsilon})$ and $O(\frac{d^4\log^2(1/\epsilon)}{\epsilon})$, respectively.*

7 Testing if a Graph is Eulerian

A graph G = (V, E) is *Eulerian* if there exists a path in the graph that traverses every edge in E exactly once. It is well known the a graph is Eulerian if and only if it is connected and all vertices have even degree or exactly two vertices have odd degree. The testing algorithm is quite straightforward. In addition to testing connectivity (as done in subsection 3.1), we sample vertices and reject whenever we see more than two vertices of odd degree. Thus we test the two properties which conjuncted together yield the desired property. However, the analysis does not reduce to showing that each of the two sub-testers is valid – as property testing of a conjunction of two sub-properties does not reduce in general to the property testing of each of the two sub-properties [GGR96]. Nonetheless, the following lemma does establish the validity of our tester.

Lemma 7.1 *Let G be a graph which is ϵ-far from the class of Eulerian graphs with maximum degree d. Then, it either has more than $\frac{\epsilon}{8}dN$ connected components, or it has more than $\frac{\epsilon}{12}dN$ vertices with odd degree.*

In other words, a graph which is ϵ-far from the class of Eulerian graphs with maximum degree d is either $\frac{\epsilon}{4}$-far from the class of connected graphs (with such degree bound) or $\frac{\epsilon}{6}$-far from the class of graphs in which all (but at most two vertices) have even degree.

Proof: Assume contrary to the claim that G has at most $\frac{\epsilon}{8}dN$ connected components, and at most $\frac{\epsilon}{12}dN$ vertices with odd degree. We now show that by adding and removing at most $\frac{\epsilon}{2}dN$ edges we can transform G into becoming a Eulerian graph.

First consider the case in which d is even, and hence all odd degree vertices have degree less than d. In such a case, we first pair all these vertices up and add an edge between every pair (using at most $\frac{\epsilon}{24}dN$ edges). Clearly, the number of connected components can only decrease in this process. At this point, all vertices have even degree, which in particular means that all (at most $\frac{\epsilon}{8}dN$) connected components either consist of a single vertex (with degree 0) or have a cycle in them. We can then remove one edge from each non-trivial component, and then connect all components in a cycle without raising the degree of the vertices above d. The total number of edge modifications is bounded by $\frac{\epsilon dN}{24} + 2 \cdot \frac{\epsilon dN}{8} < \frac{\epsilon dN}{2}$.

In case d is odd, we first remove a single incident edge from every vertex with odd degree. Since there are at most $\frac{\epsilon}{12}dN$ such vertices, at most $\frac{\epsilon}{12}dN$ edges were removed, and the number of connected components has increased by at most the same number (totaling to at most $\frac{5\epsilon}{24}dN$). However, now all vertices have even degree and we can connect the components as described above, by adding and removing at most $2 \cdot \frac{5\epsilon}{24}dN$ edges. The total number of edge modifications is bounded by $\frac{\epsilon dN}{12} + \frac{5\epsilon dN}{12} = \frac{\epsilon dN}{2}$. ■

We have thus shown that:

Theorem 7.1 *There exists a testing algorithm for the Eulerian property whose query complexity and running time are $O(\frac{\log^2(1/(\epsilon d))}{\epsilon})$.*

8 Hardness Results

In this section we present several lower bounds on the query complexity and running time required for testing various properties.

8.1 Testing Bipartiteness

A graph is said to be *bipartite* if its set of vertices can be partitioned into two disjoint sets so that there are no *violating edges*. An edge is said to be violating with respect to a given partition (V_1, V_2), if both its endpoints are either in V_1 or in V_2. In this section we show that any algorithm for testing whether a graph is bipartite has query complexity $\Omega(\sqrt{N})$. This lower bound stands in contrast to a result on testing bipartiteness which is described in [GGR96]. In [GGR96] a graph is assume to be represented by its $N \times N$ adjacency matrix, and the distance between two graphs is defined to be the fraction of entries on which their respective adjacency matrices differ. Thus, a testing algorithm for a certain graph property should distinguish between the case in which the graph has the property, and the case in which one must add and/or remove at least ϵN^2 edges in order to transform the graph into a graph that has the property. [GGR96] give an algorithm for testing bipartiteness in this model whose query complexity and running time are $\text{poly}(1/\epsilon)$. Recall that in the current paper, graphs are represented by incident lists of length d and distance is measured as the number of edge modifications divided by dN (rather than by N^2).

Theorem 8.1 *Testing Bipartiteness with distance parameter 0.01 requires* $\frac{1}{3} \cdot \sqrt{N}$ *queries.*

Proof: For any even[12] N, consider the following two families of graphs:

1. The first family, denoted $\mathcal{G}_1^N$, consists of all degree-3 graphs which are composed by the union of a Hamiltonian cycle and a perfect matching. That is, there are N edges connecting the vertices in a cycle, and the other $N/2$ edges are a perfect matching.
2. The second family, denoted $\mathcal{G}_2^N$, is the same as the first *except* that the perfect matchings allowed are restricted as follows: the distance on the cycle between every two vertices which are connected by an perfect matching edge must be odd.

In both cases we assume that the edges incident to any vertex are labeled in the following fixed manner: Each cycle edge is labeled 1 in one endpoint and 2 in the other. This labeling forms an orientation of the cycle. The matching edges are labeled 3.

Clearly, all graphs in $\mathcal{G}_2^N$ are bipartite. We next prove that almost all graphs in $\mathcal{G}_1^N$ are far from being bipartite. Afterwards, we show that a testing algorithm that performs less than $\alpha\sqrt{N}$ queries (for some constant $\alpha < 1$) is not able to distinguish between a graph chosen randomly from $\mathcal{G}_2^N$ (which is always bipartite) and a graph chosen randomly from $\mathcal{G}_1^N$ (which with high probability will be far from bipartite).

Lemma 8.1 *With probability at least* $1 - \exp(\Omega(N))$*, a graph chosen randomly in* $\mathcal{G}_1^N$ *is* 0.02*-far from the class of bipartite graphs.*

Proof: What we'll actually show is something slightly stronger: For every ordering of the vertices on the cycle, with high probability over the choice of the matching edges, the resulting graph is far from bipartite. Let us thus fix a certain ordering of the vertices on the cycle and consider all possible partitions of the graph vertices into two sets. We show that with high probability (over the choice of the matching edges) all such partitions have at least $\frac{1}{32}N$ violating edges (and since $d = 3$, this implies that the graph is ϵ-far from bipartite for $\epsilon = \frac{2\cdot(N/32)}{dN} = \frac{1}{48}$).

[12] For odd N, every graph (in both families) contains one degree-0 vertex, and the rest of the vertices are connected as in the even case.

Consider a particular partition (V_1, V_2) of V. We consider two cases: (1) There are at least $\frac{1}{32}N$ violating cycle edges with respect to (V_1, V_2). In this case we are done no matter how the matching edges are chosen. (2) There are less than $\frac{1}{32}N$ violating cycle edges. In this case we show (below) that with probability at least $1-\exp(-\frac{7}{32}N)$, over the choice of the matching edges, there are at least $\frac{1}{32}N$ violating matching edges with respect to (V_1, V_2). This will suffice since for any fixed $i \le N$, each partition which has i violating cycle edges is determined by the choice of those i violating edges. Thus there are at most $\sum_{i=0}^{N/32} \binom{N}{i} < \exp(\frac{6}{32}N)$ partitions with less than $\frac{1}{32}N$ violating cycle edges. It follows that with probability at least $1 - \exp(\frac{6}{32}N) \cdot \exp(-\frac{7}{32}N) = 1 - \exp(-\frac{1}{32}N)$ there are at least $\frac{1}{32}N$ violating matching edges with respect to each one of these partitions.

Without loss of generality, let $|V_1| \ge N/2$ and consider the following process for choosing a random matching. Starting from $j = 1$, choose an arbitrary vertex v in V_j, and match it with a randomly chosen unmatched vertex u. If the number of unmatched vertices in V_j is smaller than the number of unmatched vertices in the other side of the partition then switch side (i.e., let $j \leftarrow 3 - j$). Clearly, during this process, we always try to match a vertex from the side having more unmatched vertices. Thus, at each step we create a violating edge with probability at least $\frac{1}{2}$ (independent of the past events). Since there are $N/2$ steps, the probability that less than $\frac{1}{32}N$ violating edges are created is bounded above by $\exp(-2 \cdot (\frac{1}{2} - \frac{1}{32})^2 \cdot \frac{N}{2}) < \exp(-\frac{7}{32}N)$, as required. ■

NOTATION. Let $\mathcal{A}$ be an algorithm for testing bipartiteness using $\ell = \ell(N)$ queries. Namely, $\mathcal{A}$ is a (possibly probabilistic) mapping from *query-answer histories* $[(q_1, a_1), \ldots, (q_t, a_t)]$ to $q_{t+1} = (v_{t+1}, i_{t+1})$, for every $t < \ell$, and to $\{\textsf{accept}, \textsf{reject}\}$, for $t = \ell$. A query q_t is a pair (v_t, i_t), where $v_t \in V$ and $i_t \in \{1, 2, 3\}$, and an answer a_t is simply a vertex $u_t \in V$. We assume that the mapping is defined only on histories which are consistent with some graph. Any query-answer history of length $t-1$ can be used to define a *knowledge* graph, G^{kn}_{t-1}, at time $t-1$ (i.e., before the t^{th} query). The vertex set of G^{kn}_{t-1} contains all vertices which appear in the history (either in queries or as answers), and its edge set contains the edges between $v_{t'}$ and $a_{t'}$ for all $t' < t$ (with the appropriate labelings – $i_{t'}$ at vertex $v_{t'}$). Thus, G^{kn}_{t-1} is a labeled subgraph of the labeled graph tested by $\mathcal{A}$.

In what follows we describe two random processes, P_1 and P_2, which interact with an arbitrary algorithm $\mathcal{A}$, so that for $j \in \{1, 2\}$, P_j answers $\mathcal{A}$'s queries while constructing a random graph from $\mathcal{G}^N_j$. For a fixed $\mathcal{A}$ which uses ℓ queries, and for $j \in \{1, 2\}$, let D^A_j denote the distribution on query-answer histories (of length ℓ) induced by the interaction of $\mathcal{A}$ and P_j. We show that for any given $\mathcal{A}$ which uses $\ell \le \alpha\sqrt{N}$ queries, the statistical difference between D^A_1 and D^A_2 is $4\alpha^2$. Combining this with Lemma 8.1, Theorem 8.1 follows.

We start by defining P_1. The process has two stages. In the first stage, which goes on as long as the algorithm performs queries, the exact position of the vertices on the cycle is undetermined. However, each vertex which is introduced into the knowledge graph of the algorithm, following some query, is assigned the parity of its future position on the cycle (but this bit is NOT given to $\mathcal{A}$). That is, we think of the N positions on the cycle as being numbered from 0 to $N-1$, and a vertex which is assigned even (resp. odd) parity, will be allowed to be positioned only in even (resp. odd) cycle positions in the second stage. Thus, in this stage, the process essentially maintains the knowledge graph (which is extended according to the query-answer pairs), and keeps one additional bit per vertex. Observe that by our convention on the labeling of the edges, the knowledge graph maintained during the first stage can be viewed as "floating" (cycle) sections some of which are connected by arcs (the matching edges). In the second stage, all vertices in the final knowledge graph are positioned on the cycle randomly in a way that is consistent with the position-parity of the vertices, and so the knowledge graph edges which are labeled 1 or 2 coincide with cycle edges.

Thus the sections stop floating and are restricted to fixed positions. Finally, all vertices which do not belong to the knowledge graph are randomly positioned on the remaining cycle positions and all unmatched vertices are randomly matched.

First Stage of P_1: Starting from $t = 1$, for each query $q_t = (v_t, i_t)$ of $\mathcal{A}$, process P_1 proceeds as follows:

1. If v_t belongs to $\mathrm{G}^{\mathrm{kn}}_{t-1}$ then there are three cases:
   (a) This edge already exists in the knowledge graph (i.e., there exists an edge (v_t, u) in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ and this edge is labeled i_t at the endpoint v_t). In this case P_1 answers "u" (and the knowledge graph remains unchanged).
   (b) $i_t = 3$ and v_t is unmatched in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ (i.e., there is no edge $(v_t, \cdot)$ in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ which is labeled 3). In this case P_1 chooses a random unmatched vertex $u \in \mathrm{V}$ (where u may belong to $\mathrm{G}^{\mathrm{kn}}_{t-1}$) and answers "$u$". If u did not belong to $\mathrm{G}^{\mathrm{kn}}_{t-1}$, then it is assigned a position-parity in the natural manner. That is, let n_e be the number of vertices in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ which were assigned even parity, and let n_o be the number that where assigned odd parity. Then u is assigned even parity with probability $\frac{(N/2)-n_e}{N-(n_e+n_o)}$ and odd parity otherwise. In any case, the edge (v_t, u) is added to the knowledge graph (with label 3).
   (c) $i_t \in \{1, 2\}$ and there is no edge incident to v_t in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ which is labeled i_t. Suppose, without loss of generality, that $i_t = 1$ and v_t has even parity. Let $X_{o,2}$ be the set of vertices in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ which have odd parity, and do not have an incident edge labeled 2. Let $n_{o,2} \stackrel{\mathrm{def}}{=} |\mathrm{X}_{o,2}|$. Then P_1 first flips a coin with bias $\frac{n_{o,2}}{(N/2)-n_o+n_{o,2}}$ to decide if to choose a vertex in $X_{o,2}$. If so, it uniformly chooses a vertex in $X_{o,2}$. Otherwise, it uniformly chooses a vertex not in $\mathrm{G}^{\mathrm{kn}}_{t-1}$. Let the chosen vertex be u, Then the process answers "u", and if u does not belong to $\mathrm{G}^{\mathrm{kn}}_{t-1}$, it is assigned odd parity (i.e., parity opposite to v_t). In either case, the edge (v_t, u) is added to the knowledge graph (with label i_t at v_t).
2. If v_t does not belong to $\mathrm{G}^{\mathrm{kn}}_{t-1}$, process P_1 first assigns v_t parity as described in (1b) above, adds v_t to the knowledge graph, and next answers the query as in (1).

Second Stage of P_1: After all queries are answered, do the following:

1. Among all possible ways to embed $\mathrm{G}^{\mathrm{kn}}_{\ell}$ on the cycle, choose one uniformly, where a possible embedding of $\mathrm{G}^{\mathrm{kn}}_{\ell}$ on the cycle must satisfy the following conditions.
   (a) Every vertex is assigned a cycle position (i.e., an integer in $\{0, \ldots, N-1\}$ with parity matching the vertex's parity bit.
   (b) Vertices connected by a cycle edge in $\mathrm{G}^{\mathrm{kn}}_{\ell}$ are assigned adjacent positions on the cycle. Furthermore, if v is assigned position i on the cycle, and v has an edge labeled "1" connecting it to u in $\mathrm{G}^{\mathrm{kn}}_{\ell}$, then u must be assigned position $(i+1) \pmod N$).
2. Next, randomly position all other vertices on the cycle,
3. Finally, match all unmatched vertices randomly.

Process P_2 is the same as P_1, except when randomly matching vertices (since matched vertices must have the same position-parity). The modification to the second stage is self-evident (the unmatched vertices must be matched in a parity preserving manner). We also modify Step (1b) of the first stage – when choosing a vertex to match v_t, process P_2 only considers vertices in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ which have the same parity as v_t. Without loss of generality, assume v_t has even parity. Let $X_{o,3}$ be the set of vertices in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ which have odd parity, and do not have an incident edge labeled 3. Let $n_{o,3} \stackrel{\mathrm{def}}{=} |\mathrm{X}_{o,3}|$. Then P_2 first flips a coin with bias $\frac{n_{o,3}}{(N/2)-n_o+n_{o,3}}$ to decide if to choose a vertex in $X_{o,3}$. If so, it uniformly chooses a vertex in $X_{o,3}$. Otherwise, it uniformly chooses a vertex not in $\mathrm{G}^{\mathrm{kn}}_{t-1}$. The rest of the process, and in particular the assignment of parity to new vertices, remains unchanged.

Lemma 8.2 *For every algorithm $\mathcal{A}$ and for each $j \in \{1,2\}$, the process P_j, when interacting with $\mathcal{A}$, uniformly generates graphs in $\mathcal{G}_j^N$.*

Proof: We'll prove this by induction on the number of queries, ℓ, that $\mathcal{A}$ performs. Since every probabilistic algorithm can be viewed as a distribution on deterministic algorithms, it suffices to prove the lemma for any deterministic algorithm $\mathcal{A}$. Also note that the (accept/reject) output of the algorithm is irrelevant to the claim and hence we view the algorithm only as a mapping from histories to queries.

The base case, $\ell = 0$ is clear since the knowledge graph is empty and P_j generates a random graph in $\mathcal{G}_j^N$ from scratch. Assuming the claim is true for $\ell - 1$, we prove it for ℓ. Let $\mathcal{A}$ be an algorithm that performs ℓ queries, and let A' be the algorithm defined by stopping $\mathcal{A}$ before it asks the ℓ^{th} query. By the induction hypothesis, we know that P_j when interacting with A' uniformly generates graphs in $\mathcal{G}_j^N$. We thus need to show that the same will be true if the second stage of P_j is performed following the ℓ^{th} query of $\mathcal{A}$. We need to consider the following cases, depending on the query $q_\ell = (v_\ell, i_\ell)$ of $\mathcal{A}$. We may assume without loss of generality that the answer to the query cannot be derived from the algorithm's knowledge graph, since this would be equivalent to asking no query (in which case the knowledge graph does not change and so the distribution on P_j's output after ℓ steps is identical to its output after $\ell - 1$ steps).

1. $i_\ell = 3$, and v_ℓ belongs to the algorithm's knowledge graph, $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$. Consider first the process P_1 (when interacting with A'). The probability that P_1 matches v_ℓ (in the second stage) to any vertex (either in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ or not) is clearly independent of the exact ordering of the vertices on the cycle. Hence, by first answering this query and then performing the second stage of P_j we are only changing the order in which the final graph is constructed.

   In the case of P_2, the probability that P_2 matches v_ℓ to any vertex is still independent of the exact ordering of the vertices on the cycle, but it does depend on the parity of the vertices. In particular, assume without loss of generality that v_ℓ has even parity. Then in any possible matching done in the second stage, the only vertices in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ that v_ℓ can be matched to are vertices in $\mathrm{X}_{o,3}$ (where $\mathrm{X}_{o,3}$ is as defined in the process description). On the other hand, in any possible embedding of the vertices on the cycle, there are exactly $(N/2) - n_o$ vertices not in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ which have odd parity and thus may be matched to v_ℓ. This implies that v_ℓ is matched to some vertex in $\mathrm{X}_{o,3}$ with probability $\frac{|\mathrm{X}_{o,3}|}{|\mathrm{X}_{o,3}|+(N/2)-n_o}$, and to some vertex not in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$, with probability $\frac{(N/2)-n_o}{|\mathrm{X}_{o,3}|+(N/2)-n_o}$. Furthermore, conditioned on the event that v_ℓ is matched to a vertex in $\mathrm{X}_{o,3}$, this vertex is distributed uniformly in $\mathrm{X}_{o,3}$. Similarly, conditioned on the event that it is matched to a vertex not in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$, this vertex is uniformly distributed among vertices not in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$. But these probabilities are exactly as defined in Step (1b) or P_2.

   Therefore, for both processes the induction step holds in this case.

2. $i_\ell = 3$, and v_ℓ does not belong to $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$. This case is reduced to the previous one, provided that the parity of v_ℓ is chosen with the correct probability. In the second stage each vertex is assigned parity at random according to the proportion of missing vertices (with this parity). This is exactly the assignment rule of Step (2) in the first stage.

3. $i_\ell \in \{1,2\}$, and v_ℓ belongs to $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$. Assume, without loss of generality, that $i_\ell = 1$ and v_ℓ has even parity. Clearly, in any embedding of $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ in the cycle, v_ℓ can be adjacent to a vertex u in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ only if u belongs to $\mathrm{X}_{o,2}$ (as defined in the process). It is also clear that conditioned on the event that it is adjacent to a vertex in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$, this vertex is uniformly distributed in $\mathrm{X}_{o,2}$ (and similarly if it is not in the graph). Finally, since there should be exactly $N/2$ odd-parity vertices, and the total number of odd-parity vertices in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ is n_o, the number of odd-parity vertices not in $\mathrm{G}^{\mathrm{kn}}_{\ell-1}$ (in any ordering of the vertices on the cycle) is $(N/2) - n_o$. Thus the probability that v_ℓ is adjacent to some $u \in X_{o,2}$ is $\frac{|\mathrm{X}_{o,2}|}{|\mathrm{X}_{o,2}|+(N/2)-n_o}$, and the probability that it is adjacent to some vertex outside the knowledge graph is $\frac{(N/2)-n_o}{|\mathrm{X}_{o,2}|+(N/2)-n_o}$, which is exactly as defined by the process. Hence the induction step holds in this case.

4. $i_\ell \in \{1,2\}$, and v_ℓ does not belong to the knowledge graph. This case is reduced to the previous one, provided that the parity of v_ℓ is chosen with the correct probability. The validity of the condition was already established in Case 2.

■

Lemma 8.3 *Let $\alpha < 1$, $\ell \leq \alpha\sqrt{N}$ and $N \geq 8\ell$. Then, for every algorithm $\mathcal{A}$ which asks ℓ queries, the statistical distance between D_1^A and D_2^A is at most $4\alpha^2$. Furthermore, with probability at least $1 - 4\alpha^2$ the knowledge graph at time of termination of $\mathcal{A}$ contains no cycles.*

Recall that D_j^A denotes the distribution on query-answer histories (of length ℓ) induced by the interaction of $\mathcal{A}$ and P_j.

Proof: We assume without loss of generality that $\mathcal{A}$ does not ask queries whose answer can be derived from its knowledge graph, since those give it no new information. Under this assumption, we show that both in D_1^A and in D_2^A, the total weight of query-answer histories in which a vertex in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ is returned as an answer to the t^{th} query (i.e., there exist $t' < t$ such that $a_t = v_{t'}$ or $a_t = a_{t'}$) is at most $4\alpha^2$. In particular, this means that with probability at least $1-4\alpha^2$, the knowledge graph of $\mathcal{A}$ contains no cycles. Furthermore, in both distributions, for every history prefix, conditioned on the event that the new answer does not equal some previous query or answer, the new answer is uniformly distributed among all vertices not appearing in the history. Since $\mathcal{A}$'s queries only depend on the preceding query-answer history, the lemma follows.

There are two cases in which the event $a_t = v_{t'}$ or $a_t = a_{t'}$ might occur.

1. $i_t = 3$, and v_t is matched to a vertex in the knowledge graph $\mathrm{G}^{\mathrm{kn}}_{t-1}$. Since the number of vertices in $\mathrm{G}^{\mathrm{kn}}_{t-1}$ is at most $2(t-1)$, this event occurs with probability at most $\frac{2(t-1)}{N-2(t-1)}$ when the process is P_1, and at most $\frac{2(t-1)}{(N/2)-2(t-1)}$ when the process is P_2.

2. $i_t \in \{1,2\}$ and a_t is chosen in $\mathrm{G}^{\mathrm{kn}}_{t-1}$. According to both processes this event occurs with probability less than $\frac{2(t-1)}{(N/2)-2(t-1)} < \frac{8(t-1)}{N}$ (as $N \geq 8t$).

The probability that such an event occur in any sequence of $\alpha\sqrt{N}$ queries, is at most $\sum_{t=1}^{\alpha\sqrt{N}} \frac{8(t-1)}{N} < 4\alpha^2$. ■(Lemma 8.3 and Theorem 8.1)

8.2 Testing Whether a Graph is an Expander

The *neighbor set* of a set $\mathcal{A}$ of vertices of a graph $G = (V, E)$, denoted $\Gamma(A)$, is defined as follows:

$$\Gamma(A) \stackrel{\rm def}{=} A \cup \{u : (v, u) \in E, v \in A\}$$

A graph on N vertices is an (N, α, β)-expander if for every subset $\mathcal{A}$ of the vertices which has size at most αN, $|\Gamma(A)| \geq \beta|A|$. Let us set $\alpha = \frac{1}{4}$ and $\beta = 1.2$, and simply refer to an $(N, \frac{1}{4}, 1.2)$ expander, as an expander. Here we show that

Theorem 8.2 *Testing whether a graph is an expander, with error parameter 0.01, requires* $\frac{1}{5} \cdot \sqrt{N}$ *queries.*

Proof: Similarly to the lower bound for testing bipartiteness, we first describe two families of graphs where with extremely high probability, a graph chosen randomly in the first family is an expander, and every graph in the second family, is far from being an expander. We then describe two processes which interact with a testing algorithm while constructing a random graph in one of the families, and show that the distributions induced on the query-answer sequences are very similar. For simplicity we assume that $N \equiv 0 \pmod 8$.

Let $d = 3$. It is well known (and easy to verify) that if we randomly construct a graph by choosing d random perfect matchings to define its edge set, then with probability $1 - \exp(\Omega(N))$, the resulting graph is an expander. The first family, $\mathcal{G}_1^N$, will consist of all possible resulting graphs. A graph in the second family, $\mathcal{G}_2^N$, is constructed by first randomly partitioning the vertex set into 4 equal size subsets, and then choosing d random matchings inside each subset. Thus the four subsets are disconnected. Clearly, every graph in this family is $\frac{1}{60}$-far from being an expander, since in order to transform it into an expander we must connect each of the four subsets to at least $N/20$ vertices outside the subset. In both cases, each edge in the graph has the same label at both endpoints (i.e., corresponding to the index of the perfect matching to which the edge belongs).

The process P_1 for constructing a random graph in $\mathcal{G}_1^N$, while interacting with an algorithm $\mathcal{A}$, is completely straightforward. Let $q_t = (v_t, i_t)$ be $\mathcal{A}$'s $t^{\rm th}$ query. If the answer a_t is determined by the current knowledge graph, $G_{t-1}^{\rm kn}$, then P_1 answers accordingly. Otherwise, it chooses a random vertex u which does not have an incident edge labeled i_t, answers "u", and adds the edge (and possibly the vertex) to the knowledge graph. When the interaction with $\mathcal{A}$ ends, P_1 randomly completes all d matchings.

Process P_2 is somewhat more complex. It maintains four subsets of vertices and coordinates its choice of matching edges with these growing subsets.

- Whenever algorithm $\mathcal{A}$ makes a query of the form (v, i) where v is not in the current knowledge graph, P_2 assigns it a *subset-id* in $\{1, 2, 3, 4\}$ with probability proportional to the number of vertices missing in each subset (P_2 starts with all subsets being empty). Specifically, let n_s be the number of vertices with subset-id s in the current knowledge graph, for $s = 1, 2, 3, 4$. Then the new vertex is assigned subset-id s with probability $\frac{(N/4)-n_s}{N-(n_1+n_2+n_3+n_4)}$. The query is subsequently processed as follows.

- To answer a query (v, i) when v is already in the current knowledge graph, P_2 matches it to either a vertex already assigned to the same subset as v or to an unassigned vertex. Specifically, suppose that v is already assigned to the $s^{\rm th}$ subset, and let $X_{s,i}$ denote the set of vertices which are assigned to the $s^{\rm th}$ subset but do not have an incident edge labeled i. Then with probability $\frac{|X_{s,i}|-1}{(N/4-n_s)+(|X_{s,i}|-1)}$ process P_2 matches v to a uniformly chosen vertex, u, in $X_{s,i} \setminus \{v\}$. Otherwise, P_2 matches v to a uniformly chosen vertex, u, which does not belong to the current knowledge graph, and assigns u to the s^{th} subset. In both cases P_2 answers with the chosen vertex u, and the knowledge graph is augmented with the edge (v, u) labeled i.

It is easy to verify that for both processes the distribution on the generated graphs is uniform in the respective graph family. Similarly to the bipartite lower bound, it remains to show that for any (not too long) query-answer history, the probability that we get an answer a_t which is a vertex in the knowledge graph (and not a uniformly distributed new vertex) is small. But this is easy to see. In the case of P_1, such a vertex is chosen following the t^{th} query, with probability at most $\frac{2t}{N-2t}$. In the case of P_2, such a vertex is chosen with probability at most $\frac{2t}{(N/4)-2t}$. The probability that such an event occurs in any sequence of $\alpha\sqrt{N}$ queries, is at most $\sum_{t=1}^{\alpha\sqrt{N}} \frac{8t}{N-8t}$ which is at most $8\alpha^2$, for every $N \geq 256$. ■

8.3 On Testing MaxSNP Problems

It should come with little surprise that we cannot efficiently test some properties of bounded degree-graphs which are MaxSNP-complete (i.e., when restricted to bounded-degree graphs). To see why, we need to be a bit more formal.

Consider for example the class $\mathcal{C}_d^\rho$ of graphs with maximum degree d having a vertex cover of size ρN, for some constant $\rho > 0$. Suppose that $\mathcal{A}$ is a property tester for $\mathcal{C}_d^\rho$ as in Definition 2.1. Suppose we have an algorithm $\mathcal{A}$ which on input ϵ and d, and access to a graph with degree bounded by d, the algorithm accepts (with high probability) any graph in $\mathcal{C}_d^\rho$ but rejects (w.h.p.) any N-vertex graph (of degree $\leq d$) which requires modification of ϵdN edges in order to be in $\mathcal{C}_d^\rho$. We observe that it suffices to consider the number of edges omitted in the modification process, and that the number of omitted edges can be related to an increase in the vertex cover. Specifically,

Claim 8.4 *Suppose that $\mathcal{A}$ is a property tester for $\mathcal{C}_d^\rho$. Then $\mathcal{A}$ distinguishes between N-vertex graphs (of degree at most d) having vertex cover of size $\rho \cdot N$ and similar graphs having no vertex cover of size $(\rho + \epsilon d) \cdot N$.*

Since distinguishing the two cases is NP-Hard for some constants d, ϵ and ρ [ALM+92, PY91], we cannot expect $\mathcal{A}$ to have "reasonable" (e.g., polynomial in N) complexity.

Proof: By definition, the former graphs are in $\mathcal{C}_d^\rho$. It remains to see that N-vertex graphs having no vertex cover of size $(\rho + \epsilon d) \cdot N$ require the modification of at least ϵdN edges in order to put them in $\mathcal{C}_d^\rho$. Suppose that it suffices to omit m edges from a graph G in order to obtain a graph G′ in $\mathcal{C}_d^\rho$ (we don't care if edges were added in the process).[13] Then taking the ρN-vertex-cover of G′ and at most one endpoint of each of the m edges omitted from G, results in a vertex cover of G having size at most $\rho N + m$. Thus, we have $m \geq \epsilon dN$. ■

Next, we consider the class $\mathcal{D}_d^\rho$ of graphs with maximum degree d having a dominating set of density ρ. We observe that it suffices to consider the number of edges which need to be added to put the graph in $\mathcal{D}_d^\rho$. Specifically,

[13] Actually, without loss of generality we may assume that no edges were added as they only make the task of covering harder...

Claim 8.5 *Suppose that $\mathcal{A}$ is a property tester for $\mathcal{D}_d^\rho$. Then $\mathcal{A}$ distinguishes between N-vertex graphs (of degree at most d) having no dominating set of size $\rho \cdot N$ and similar graphs having dominating set of size $(\rho + \epsilon d) \cdot N$.*

Again, since distinguishing the two cases is NP-Hard for some constants d, ϵ and ρ [ALM+92, PY91], we cannot expect $\mathcal{A}$ to have "reasonable" complexity.

Proof: Again, the former graphs are in $\mathcal{D}_d^\rho$, and it remains to see that N-vertex graphs having no dominating set of size $(\rho + \epsilon d) \cdot N$ require the modification of at least ϵdN edges in order to put them in $\mathcal{D}_d^\rho$. Suppose that it suffices to add m edges to a graph G, with maximum degree d, in order to obtain a graph G$'$ in $\mathcal{D}_d^\rho$ (we don't care if edges were omitted in the process).[14] Let S' be a dominating set of size ρN of G$'$. Then S' dominates all but at most m vertices in G (i.e., all vertices dominated in G$'$ except for those which are dominated due to the edges added to G). Adding these vertices to S' we obtain a dominating set of size $|S'| + m$ of G, and thus $m \geq \epsilon dN$. ■

We conclude by proving a lower bound on the query complexity of testers for the Vertex Cover Property, $\mathcal{C}_d^\rho$. Specifically,

Proposition 8.3 *Let $d = 3$, $\rho = 0.5$ and $\epsilon = 0.01$. Then testing whether a 3-regular N-vertex graph belongs to $\mathcal{C}_d^\rho$ or is ϵ-far from it requires $\Omega(\sqrt{N})$ queries.*

Proof: We use the families $\mathcal{G}_1^N$ and $\mathcal{G}_2^N$ presented in Subsection 8.1. By combining Lemmas 8.2 and 8.3, an algorithm which makes $o(\sqrt{N})$ queries can not distinguish graphs uniformly chosen in $\mathcal{G}_1^N$ from graphs uniformly chosen in $\mathcal{G}_2^N$. It is easy to see that graphs in $\mathcal{G}_2^N$ have a vertex cover of size $N/2$ (e.g., all vertices with odd locations on the cycle). It remains to show that, with very high probability, a graph chosen uniformly in $\mathcal{G}_1^N$ has no vertex cover of size $0.51 \cdot N$.

Consider an arbitrary subset, U, of vertices which cover all cycle edges. Then each vertex not in U must be adjacent (on the cycle) to vertices in U. It follows that the number of possible subsets of size $0.51N$ which cover the cycle edges is at most

$$\binom{0.51N}{0.49N} < 2^{N/6}$$

On the other hand, for every fixed U as above, the probability that U covers the matching edges is upper bounded by

$$\prod_{i=0}^{0.01N} \binom{0.51N - i}{N - 2i} < 2^{-0.4N}$$

We conclude that the probability that a graph chosen uniformly in $\mathcal{G}_1^N$ has a vertex cover of size $0.51 \cdot N$ is exponentially vanishing in N. The lemma follows. ■

Acknowledgments

We thank Yefim Dinitz, Shimon Even, David Karger for helpful discussions.

[14] Here we cannot assume that the modification of G into G$'$ consists only of the addition of edges, since we may be forced to omit edges in order to satisfy the degree bound. Nevertheless, this fact does not effect the proof.

References

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and intractability of approximation problems. In *Proceedings of the Thirty-Third Annual Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[AS92] S. Arora and S. Safra. Probabilistic checkable proofs: A new characterization of NP. In *Proceedings of the Thirty-Third Annual Symposium on Foundations of Computer Science*, pages 1–13, 1992.

[Ben95] A. Benczur. A representation of cuts within 6/5 times the edge connectivity with applications. In *Proceedings of the Thirty-Sixth Annual Symposium on Foundations of Computer Science*, pages 92–101, 1995.

[BFL91] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1(1):3–40, 1991.

[BFLS91] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 21–31, 1991.

[BGS95] M. Bellare, O. Goldreich, and M. Sudan. Free bits, pcps and non-approximability – towards tight results. In *Proceedings of the Thirty-Sixth Annual Symposium on Foundations of Computer Science*, pages 422–431, 1995. Full version available from *ECCC*, `http://www.eccc.uni-trier.de/eccc/`.

[BLR93] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47:549–595, 1993.

[DKL76] E. A. Dinic, A. V. Karazanov, and M. V. Lomonosov. On the structure of the system of minimum edge cuts in a graph. *Studies in Discrete Optimizations*, pages 290–306, 1976. In Russian.

[DW95] Y. Dinitz and J. Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. Technical Report #871, Technion, Department of Computer Science, 1995.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, 1979.

[FGL+91] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *Proceedings of the Thirty-Second Annual Symposium on Foundations of Computer Science*, pages 2–12, 1991.

[Gab91] H. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the Thirty-Second Annual Symposium on Foundations of Computer Science*, pages 812–821, 1991.

[Gab95] H. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995.

[GGR96] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. In *Proceedings of the Thirty-Seventh Annual Symposium on Foundations of Computer Science*, pages 339–348, 1996.

[GLR+91] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 32–42, 1991.

[Hås96] J. Håstad. Testing of the long code and hardness for clique. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 11–19, 1996.

[Kar93] D. Karger. Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 21–30, 1993.

[Kar95] D. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Stanford University, 1995. Available from `http://theory.lcs.mit.edu/~karger`.

[Kur30] K. Kuratowski. Sur le probleme des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.

[NGM90] D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge-connectivity. In *Proceedings of the Thirty-First Annual Symposium on Foundations of Computer Science*, pages 698–707, 1990.

[NI96] H. Nagamochi and T. Ibaraki. Deterministic $\tilde{O}(nm)$ time edge-splitting in undirected graphs. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 64–73, 1996.

[Pou92] J. A. La Poutre. Maintenance of triconnected components of graphs. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 354–365, 1992. Springer-Verlag Lecure Notes in Computer Science 623.

[PY91] C.H. Papadimitriou and M. Yanakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[RS96] R. Rubinfeld and M. Sudan. Robust characterization of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2):252–271, 1996.

[WN87] T. Watanabe and A. Nakamura. Edge-connectivity augmentation problems. *Journal of Computer and System Sciences*, 35:96–144, 1987.

A Background on Edge-Connectivity

In this appendix we recall some known facts regarding the structure of the k-edge-connected classes of a $(k-1)$-connected graph. Whereas the structure of the 2-classes of a connected graph is well-known and relatively simple (cf., [Eve79]), the (k-connected class) structure of $(k-1)$-connected graphs becomes slightly more complex when $k \geq 3$. We thus refrain from describing in detail this structure and merely state the facts which we need. The interested reader is referred to [DW95] for more details. We stress that the graphs below are not necessarily simple; that is, parallel edges are allowed.

Fact A.1 (cf., [DW95]): *Let $k > 1$ be an integer and* G *be a $(k-1)$-connected graph. Then there exists an auxiliary graph,* T_{G}*, that is a tree such that:*

- *Each k-connected class in* G *corresponds to a unique node in* $\mathrm{T_G}$.
- *In addition to nodes corresponding to k-connected classes, there are two types of auxiliary nodes:* empty *nodes, and* cycle *nodes (the latter exist only for odd k). The neighbors of a cycle node in* $\mathrm{T_G}$ *are said to* belong *to a common cycle, and we associate a cyclic order with them. (Any two cycles can have at most one common node.)*
- *All leaves of the auxiliary tree* $\mathrm{T_G}$ *correspond to k-connected classes of* G*. Furthermore, there are exactly $k-1$ edges (in* G*) going out from each of these classes.*

For example, when $k = 2$, all nodes of the auxiliary tree correspond to 2-classes, and the edges in the auxiliary tree correspond to graph edges which are known as *bridges*. Bridges are edges connecting vertices in different 2-classes of the graph, and their removal disconnects the graph. In the case of $k = 3$, the auxiliary tree includes cycle nodes (but no empty nodes). If $\mathrm{C}_1 \ldots, \mathrm{C}_\ell$ are neighbors of a cycle node Cy, then this means that there is a single graph edge between some vertex in C_i and some vertex in C_{i+1}, (for every $i < \ell$) and an edge between a vertex in C_ℓ and a vertex in C_1.

Before stating the next lemma we need to define the notion of *squeezing a cycle*. Let Cy be a cycle node in $\mathrm{T_G}$, and let its children be $\mathrm{C}_1, \ldots, \mathrm{C}_t$ (where their indices corresponds to their ordering around the cycle). Then the result of *squeezing* Cy at C_i and C_j is the merging of C_i and C_j into a new node C_k, with one of the following changes to the cycle:

1. In case C_i and C_j are adjacent on the cycle, then we have two subcases. If $t > 3$ then the merged node C_k is connected by a single edge to the cycle node Cy (and all other nodes belonging to the cycle remain that way); If $t = 3$ (i.e., there was only one additional node on the cycle), then Cy is removed, and the additional node is connected by a tree edge to C_k.
2. In case C_i and C_j are separated by at least one node on the cycle then we have three subcases.
   (a) If $t = 4$ (i.e., C_i and C_j are separated by a single node in each cycle direction), then we put a tree edge between each of these intermediate nodes and C_k, and the cycle disappears.
   (b) Otherwise ($t > 4$). If C_i and C_j are separated by a single node, then we put a tree edge between this node and C_k, and C_k belongs to a single cycle with all the rest of the nodes which were previously on the cycle.
   (c) Otherwise at least two nodes separate C_i and C_j in each direction. Then we get two cycles, where C_k belongs to both, and the other nodes are partitioned among the cycles according to their relative position with respect to C_i and C_j.

Lemma A.2 (cf., [DW95]): *Let* G *be a $(k-1)$-connected graph, and T_{G} be its auxiliary tree. Suppose that we augment* G *by an edge with endpoints in the k-connected classes C_1 and C_2, respectively. Then the classes residing on the simple path between C_1 and C_2 in T_{G} form a k-connected class in the augmented graph, and all classes in* G *which do not reside on the path remain distinct k-classes in the augmented graph. In case the path passes through nodes* C_i *and* C_j *which belong to the same cycle* Cy*, then* Cy *is squeezed at* C_i *and* C_j.

A related lemma which we need follows. We note that this lemma can be proven (*private communication with Y. Dinitz, December 1996*) using the Circumference Theorem in [DKL76], but we provide a direct proof for completeness. In what follows, when we refer to an edge as *being in a class* we mean that it connects two vertices belonging to the class.

Lemma A.3 *Let* G *be a* $(k-1)$*-connected graph,* T_{G} *be its auxiliary tree, and* C_1*,* C_2 *two* (k-connected) *classes of* G *each containing at least one edge. Suppose that we omit a single edge from each* C_i *and add two edges so to maintain the vertex degrees of* G*; Specifically, if the edges* (u_1, v_1) *and* (u_2, v_2) *were omitted from* C_1 *and* C_2 *respectively, then we either add the edges* (u_1, u_2) *and* (v_1, v_2)*, or the edges* (u_1, v_2) *and* (v_1, u_2)*. As a result, the classes residing on the simple path between* C_1 *and* C_2 *in* T_{G} *form a* k*-connected class in the augmented graph, and all classes in* G *which do not reside on the path remain distinct* k*-classes in the augmented graph.*

Proof: Let $I_1, ..., I_t$ be the (intermediate) k-classes residing on the path between C_1 and C_2 in the tree T_{G}. (We do not exclude the case $t = 0$.)

Consider what happens when we omit the edge (u_i, v_i) from C_i. Invoking Lemma A.2, we observe that C_i may break into a path of subclasses, denoted $\mathrm{C}_i^1, ..., \mathrm{C}_i^{q_i}$, so that the vertex u_i resides in C_i^1, and vertex v_i resides in $\mathrm{C}_i^{q_i}$. Thus, the I_j's and the C_i^j's reside on a tree (i.e., a subtree of the modified auxiliary graph) so that the only leaves are among the "extreme" C_i^j's (i.e., C_1^1, $\mathrm{C}_1^{q_1}$, C_2^1, and $\mathrm{C}_2^{q_2}$). (We do not exclude the case $q_i = 1$. In this case $\mathrm{C}_i = \mathrm{C}_i^1$ and the above subtree has less than four leaves.)

At this point, the only case in which we must practice caution in choosing which two edge to add, is the case in which C_1^1, $\mathrm{C}_1^{q_1}$, C_2^1, and $\mathrm{C}_2^{q_2}$ all belong to the same cycle. In such a case assume that the above is in fact their ordering around the cycle. Then it is essential that we add the edges (u_1, u_2) and (v_1, v_2) (i.e., connecting C_1^1 to C_2^1 and $\mathrm{C}_1^{q_1}$ to $\mathrm{C}_2^{q_2}$) in a *crossing* fashion, so as to insure that the two invocation of Lemma A.2 will cause the collapse of the four classes into one class. In all other cases, we may invoke Lemma A.2 twice, either with the addition of the edges (u_1, u_2) and (v_1, v_2), or with the edges (u_1, v_2) and (v_1, u_2), resulting in the collapse of the entire subtree to one class. The lemma follows. ■

Using Lemmas A.2 and A.3, we get.

Lemma A.4 *Let* G *be a* $(k-1)$*-connected graph, whose auxiliary graph,* T_{G}*, has* L *leaves. Then by removing and adding at most* $4L$ *edges to* G *we can transform it into a* k*-connected graph* G'*. Furtheremore, suppose that the maximum degree of* G *is* d *then the maximum degree of* G' *is upper bounded by* $\max\{d, k\}$ *if either* $d > k$ *or* dN *is even, and by* $k+1$ *otherwise.*

We note that there might be a way to save a constant factor in the number of edges added and removed from G when transforming it into a k-connected graph by using a result of Naor, Gusfield and Martel [NGM90]. They give an algorithm for optimally increasing the edge connectivity of a graph. However, they do so by always adding edges, without maintaining a bound on the degree of the graph, and hence it is not clear if their techniques can be applied in our, bounded degree, case.

Proof: We first use Lemma A.2 to collapse all leaves in T_{G} which correspond to singleton classes. These vertices have degree $k-1$ and so we can match them in pairs and add a single edge between each pair. At this point we may be left with a single unmatched vertex/leaf, which we deal with later. Call the resulting graph G_1 and its auxiliary tree T_1. The number of leaves in T_1 is at most $L - i$, where i is the number of pairs matched above. All leaves in T_1 (except for the possible singleton) can be now collapsed using Lemma A.3. The number of edge modifications in this stage is at most $4(L-i)$. The resulting graph, G_2, has degree at most $d' \stackrel{\mathrm{def}}{=} \max\{d, k\}$. In case G_2 is k-connected we are done.

Otherwise, G_2 consists of a singleton which is connected to a k-connected class containing all other vertices. In case some vertex in the large class has degree lower than d' we connect it to the singleton and conclude as per Lemma A.2. Otherwise (i.e., all vertices in the large class have degree d'), we need to distinguish two subcases. In case $k < d'$ we simply omit one edge internal to the large class and connect its endpoints to the singleton. It can be seen that this makes the graph k-connected and that all vertices have degree at most d'. Finally, if $d' = k$ a parity argument shows that both d' and N must be odd (as otherwise the sum of degrees is odd). In this case we are allowed to add an edge and increase the degree of the resulting graph to $d' + 1 = k + 1$. ■

B Background on Vertex-Connectivity

In this appendix we recall some known facts regarding the structure of the k-vertex-connected classes of a $(k-1)$-connected graph for $k = 2, 3$, and derive some new facts needed for our testing algorithms. When we refer to k–classes, we think of k–vertex–connected classes of size at least $k+1$. We use the convention that every pair of vertices which are connected by an edge and do not both belong to the same k–class, form a class of their own which we refer to as an *edge*–class. In the case of $k = 3$, we choose to view triplets of vertices which form a clique and do not all belong to a common k–class as a *cycle* of edge–classes (see below).

For $k = 2$, each class corresponds to a *component* which is the subgraph induced by the vertices in the class. The components corresponding to 2-vertex-connected classes are referred to as 2-vertex-connected components, and those corresponding to edge classes, as edge components. The (at least) two paths that connect a pair of vertices which belong to the same 2-vertex-connected class use only vertices and edges which reside in the corresponding component. Hence, these components are 2-vertex-connected subgraphs. For $k = 3$ there also exists such a correspondence between classes and components only it is slightly more involved since the components include edges which are not in the original graph. We return to this issue later. For simplicity, from now on we refer to k-vertex-connected classes (components) as k-classes (components).

B.1 The 2-classes (components) of a connected graph

Given a connected graph G we can define an auxiliary graph $\mathrm{T_G}$ whose nodes are 2-vertex-connected classes, edge-classes, and separating vertices of G. For every class C in G (where C is either a 2-class or an edge-class), we have an edge in $\mathrm{T_G}$ between C and each of the separating vertices it contains. Since G is connected and every pair of classes in G have at most one separating vertex in common, $\mathrm{T_G}$ is a tree. Since 2-classes directly correspond to 2-components, we use the two terms interchangibly (depending on whether we want to discuss sets of vertices or subgraphs (sets of edges)).

Lemma B.1 *Let* G *be a connected graph,* C_1 *and* C_2 *two 2-classes in* G*, and* $\mathrm{X}^1, \mathrm{X}^2, \ldots, \mathrm{X}^{\ell-1}, \mathrm{X}^\ell$ *the sequence of classes on the path connecting* C_1 *and* C_2 *in* $\mathrm{T_G}$*, where* $\mathrm{X}^1 = \mathrm{C}_1$ *and* $\mathrm{X}^\ell = \mathrm{C}_2$*. Suppose that we augment* G *by an edge* (v_1, v_2) *where* $v_1 \in \mathrm{C}_1$*,* $v_2 \in \mathrm{C}_2$*, and such that neither vertex separates* $\mathrm{C} \stackrel{\mathrm{def}}{=} \cup \mathrm{X}^i$ *in* G*. Then* C *is a 2-class in the augmented graph* G'*, and all classes in* G *which do not reside on the path remain distinct 2-classes in* G'*.*

Proof: We need to show that for every vertex v, the removal of v from G' does not disconnect any pair of vertices in C. Clearly, by definition of C (and the properties of the auxiliary graph), if $v \notin \mathrm{C}$ then it cannot separate C (in G or G'). Furthermore, if v does not separate C in G then it does not separate C in G'. Thus, let v be a separating vertex of G which belongs to C, and let X^j be the class it belongs to. Note that by our assumption on v_1 and v_2, v can be neither of them. Consider a pair of vertices, u_1 and u_2 in C which are disconnected in G by the removal of v. We now show that they are not separated by v in G$'$. (Clearly, pairs which are not separated by v in G are not separated by it in G$'$). Let X^{j_1} and X^{j_2} be the classes they belong to respectively, and without loss of generality, $j_1 < j_2$ and $j < j_2$ (if a vertex belongs to more than one class than we choose the one with the smaller index). After the removal of v, the subgraphs induced by $X^1, \ldots, X^j \setminus \{v\}$ and $X^{j+1} \setminus \{v\}, \ldots, X^\ell$ respectively, are each connected. Hence there still exists a path from u_1 to v_1 and a path from u_2 to v_2. Since we added the edge (v_1, v_2), u_1 and u_2 are connected.

As for pairs of vertices u_1, u_2, that are not both in C, it is not hard to verify based on the definitions of T_G and C, that the addition of the edge (v_1, v_2) cannot increase their connectivity, since they are still separated by the same separating vertices. ■

By applying the reverse operation to that described in Lemma B.1 (i.e., removing an edge), we get the following corollary.

Corollary B.2 *Let* G *be a connected graph, and* C *a 2-class in* G*. Let v_1 and v_2 be any two vertices in* C *that are connected by an edge in* G*. Assume we remove (v_1, v_2) from* G*. Then the resulting graph* G$'$ *is connected, and the vertices in* C *belong to classes $X^1, \ldots, X^\ell$ in* G$'$ *such that $v_1 \in X^1$, $v_2 \in X^\ell$, and $X^1, \ldots, X^\ell$ form a simple path in $T_{G'}$. Furthermore, all other classes in* G *are classes in* G$'$*, and there are no other classes in* G$'$*. (Note that the case in which* C *remains a 2-class in* G$'$ *is a special case of the above).*

By applying Corollary B.2 and Lemma B.1 we get:

Lemma B.3 *Let* G *be a connected graph, T_G its auxiliary graph, and C_1, C_2 be two 2-classes in* G*. Then there exists an edge $e_1 = (u_1, v_1)$ between two vertices in C_1 and an edge $e_2 = (u_2, v_2)$ between vertices in C_2, for which the following is true. If we remove e_1 and e_2, and add an edge between v_1 and v_2 then the classes residing on the simple path between C_1 and C_2 in T_G form a 2-class in the modified graph* G$'$*.*

Proof: Let $Y^1, Y^2, \ldots, Y^{\ell-1}, Y^\ell$ be the sequence of classes on the path connecting C_1 and C_2 in T_G where $Y^1 = C_1$ and $Y^\ell = C_2$. Let u_1 be the separating vertex common to C_1 and Y^2, let u_2 be the separating vertex common to C_2 and $Y^{\ell-1}$, and let $e_i = (u_i, v_i)$ be any edge between u_i and a vertex v_i in C_i. Note that u_1 and u_2 coincide in case C_1 and C_2 are neighbors. Consider first the removal of e_1, and denote the resulting graph by G_1. By Corollary B.2, in G_1, the vertices belonging to C_1 are possibly divided (in a non-disjoint manner) into classes $X_1^1, \ldots, X_1^{\ell_1}$ where $v_1 \in X_1^1$ and $u_1 \in X_1^{\ell_1}$ (and the other classes in G remain unchanged in G_1). Since u_1 belongs to Y^2 as well, we have a path in T_{G_1} between X_1^1 and Y^2. We next remove e_2 from G_1, and let the resulting graph be denoted G_2. Then in G_2 the vertices in C_2 are possibly divided into classes $X_2^1, \ldots, X_2^{\ell_2}$ where $v_2 \in X_2^1$ and $u_2 \in X_2^{\ell_2}$. Thus the auxiliary graph T_{G_2} has the following path from X_1^1 to X_2^1:

$$X_1^1, \ldots, X_1^{\ell_1}, Y^2, \ldots, Y^{\ell-1}, X_2^{\ell_2}, \ldots, X_2^1$$

Note that by the above, neither v_i separates the union of the classes on the path. When we add the edge between v_1 and v_2 then by Lemma B.1, we get a new class

$$C \;=\; \cup_{i=1}^{\ell_1} X_1^i \bigcup \cup_{i=2}^{\ell-1} Y^i \bigcup \cup_{i=1}^{\ell_2} X_2^i \;=\; \bigcup_{i=1}^{\ell} Y^i$$

as required. ■

Lemma B.4 *Let* G *be a connected graph whose auxiliary graph,* T_G *has* L *leaves. Then by removing and adding at most* $3L$ *edges to* G *we can transform it into a 2-connected graph* G'. *Furtheremore, the maximum degree of vertices in* G' *is at most the maximum degree in* G.

Proof: Let L_1 be the number of edge classes which are leaves of T_G and let L_2 be the number of 2-classes which are leaves of T_G. Since G is connected, its maximum degree is at least 2, and its only degree-1 vertices belong to edge classes which are leaves. Assume first that L_1 is even. In such a case we pair the edge leaves, and add an edge between the degree-1 vertices in each pair (raising their degree to 2). By Lemma B.1, each pair of edge classes now belongs to a single class (which possibly includes other vertices). The auxiliary graph (tree) $T_{G'}$ of the resulting graph G' has at most $L_1/2 + L_2$ 2-class leaves (and no edge leaves). Applying Lemma B.3 at most $L_1/2 + L_2$ times we can merge all classes of G' into a single 2-class by removing and adding at most $3(L_1/2 + L_2)$ edges in and between the leaves of $T_{G'}$. Note that in the process described in Lemma B.3 we do not increase the degree of any vertex.

In case L_1 is odd, we can pair all but one edge class. Let this class be $\{u, v\}$ where u is the separating vertex which also belongs to a neighbor 3-class C. Let v' be a neighbor of u in C, then it is not hard to verify that by removing the edge (u, v') and adding the edge (v, v'), the set $C \cup \{v\}$ becomes a new 3-class. ■

B.2 The 3-classes (components) of a 2-vertex-connected graph

Here the structure becomes more complicated and we refer the reader to [Pou92]. First we describe how to decompose a 2-connected graph G into components of two types: 3-components (with at least 3 vertices) and cycles. Slightly differently from the case of two-components of a connected graph which are subgraphs of the graph, here the components contain additional edges (which "stand for" paths using vertices outside the component). However, it is still true that the set of vertices of each 3-component is a 3-class of the graph, and with slight abuse of terminology, we shall sometimes interchange between the two terms. We later show how to construct an auxiliary graph (which is a tree) whose nodes correspond to 3-components, cycles, edge components and separating pairs.

If G is 3-vertex-connected then it consists of a single 3-component. If G is a simple cycle, then it consists of a single cycle component. Otherwise there must exist at least one separating pair of vertices in G that are 3-connected. Let v_1 and v_2 be such a pair which we call a *block*. The removal of v_1 and v_2 induces a partition of $V \setminus \{v_1, v_2\}$ into disjoint subsets $V_1, \ldots, V_\ell$ that are maximal sets which remain connected after the removal of v_1, v_2. For each V_j, let E_j be the set of edges that have endpoints in V_j. Since $\{v_1, v_2\}$ is a separating pair, the sets E_j are disjoint. For each j let G_j be the graph whose vertex set is $V_j \cup \{v_1, v_2\}$ and whose edge set is the union of E_j with *two* edges between v_1 and v_2. A single edge between v_1 and v_2 would suffice to ensure that for each G_j and for every pair of vertices $y_1 \in V_j$ and $y_2 \in V_j \cup \{v_1, v_2\}$, y_1 and y_2 have the same vertex-connectivity in G_j as in G. However, we allow for multiple edges between v_1 and v_2 so that they remain 3-vertex-connected in G_j. Each G_j is either 3-connected, or it is a cycle, or it contains a block (a separating 3-connected pair).

We thus continue recursively to decompose $G_1, \ldots, G_\ell$ as described above, with the minor exception that whenever we obtain a non-simple cycle (due to the addition of multiple edges), we turn it into a simple cycle. The resulting decomposition is independent of the order of separating pairs chosen. Since the decomposition process induces a partition on E,[15] For an example of the

[15]By the above, a block in G (or more generally in some G_j defined by the recursive decomposition) induces a

decomposition process, see Figure 1.

The auxiliary graph T_G (which we shall also refer to as the *decomposition* graph) is constructed using the above decomposition process as follows. In case G is 3-connected then T_G contains a single node corresponding to its single 3-component (a *3-component node*). In case G is a cycle, then T_G contains one node corresponding to its cycle component (a *cycle node*), one node for each edge component on the cycle (an *edge node*), and an edge between the cycle node and each one of the edge nodes. (Recall that we use the convention that edges which do not belong to any 3-component are considered as separate edge components). Otherwise, let $\{v_1, v_2\}$ be the block according to which we decompose G. Then we have a node $\{v_1, v_2\}$ in T_G corresponding to this block (a *block node*). We next recursively construct the decomposition trees of $G_1, \ldots, G_\ell$ and connect them to $\{v_1, v_2\}$ as follows. For each G_j, if (v_1, v_2) is an edge node (i.e., part of a cycle), then we identify it with the node $\{v_1, v_2\}$ (that is, we discard the node from T_{G_j} and put an edge between $\{v_1, v_2\}$ and its cycle node neighbor). Otherwise, v_1 and v_2 must belong to a 3-component C, and we put an edge between $\{v_1, v_2\}$ and the node corresponding to C. See Figure 2 for an example of a decomposition tree (of the graph depicted in Figure 1).

As noted previously, T_G is a tree. While a vertex v in G may belong to several components and blocks which have corresponding nodes in the tree, these nodes induce a connected subgraph (subtree) of T_G, which we denote by $S_G(v)$. If v_1 and v_2 belong to a common 3-components and/or bar (that is, $S_G(v_1)$ and $S_G(v_2)$ are not disjoint) then by definition, they are 3-connected. Otherwise we denote by $P_G(v_1, v_2)$ the path connecting $S_G(v_1)$ and $S_G(v_2)$ in T_G (since T_G is a tree, this path is well defined). By definition of T_G, (and the 2-connectivity of G), v_1 and v_2 have two vertex disjoint paths that pass only through vertices that belong to bars and 3-components on $P_G(v_1, v_2)$. Furthermore:

Lemma B.5 (cf., [Pou92]) *Let* G *be a 2-connected graph, and let* v_1 *and* v_2 *be any two vertices in* G *which are not 3-connected. Assume we augment* G *with an edge between* v_1 *and* v_2*. Then* v_1*,* v_2*, together with all the vertices in the 3-components and bars on the path* $P_G(v_1, v_2)$*, form a new 3-class in the augmented graph* G′*. All classes in* G *which do not reside on the path are classes in* G′ *and there are no other classes in* G′*.*

Proof: Let $X^1, \ldots, X^\ell$, be the sequence of 3-components and bars on $P_G(v_1, v_2)$, and let C be the union of v_1,v_2 and the vertices belonging to $X^1, \ldots, X^\ell$. From the definitions of T_G and C, any two vertices that do not both belong to C cannot separate C (neither in G not in G′). Also, any pair of vertices that does not separate C in G cannot separate C in G′. We can thus restrict our attention to separating pairs in C (with respect to G), where by definition of C all such pair are blocks in G. Let X^j be a block whose vertices belong to C, and let y_1 and y_2 be two vertices that X^j separates in G. Then the removal of X^j separates G into at least 2 connected subgraphs: one which contains v_1 and all vertices in $X^1, \ldots, X^{j-1}$ (excluding the two vertices in X^j), and the other which contains v_2 and all vertices in $X^{j+1}, \ldots, X^\ell$ (excluding the two vertices in X^j). Furthermore, y_1 belongs to one of these subgraphs (without loss of generality, let it be the first) and y_2 belongs to the second subgraph. Thus y_1 is connected to y_2 in G′ via the path $y_1 — v_1 - v_2 — y_2$, where $y_1 — v_1$ passes in the first subgraph and and $v_2 — y_2$ in the second subgraph. Therefore, all vertices in C are 3-connected in G′.

As for pairs of vertices y_1, y_2, that are not both in C, it is not hard to verify based on the definitions of T_G and C, that the addition of the edge (v_1, v_2) cannot increase their connectivity.

partition on all edges of E (E_j) except the edge that might exist between the block vertices. In such a case we arbitrarily place this edge in one of the partition's subsets.

■

By applying the reverse operation to that described in Lemma B.5 (i.e., removing an edge), we get the following corollary.

Corollary B.6 *Let* G *be a 2-connected graph, and* C *a 3-class in* G*. Let* v_1 *and* v_2 *be two vertices in* C *which are connected by an edge in* G*. Assume we remove* (v_1, v_2) *from* G*. Then the resulting graph* G' *is 2-connected, and either* C *remains a 3-class in* G' *or the vertices in* $\mathrm{C} \setminus \{v_1, v_2\}$ *belong to the 3-components and bars* $\mathrm{X}^1, \ldots, \mathrm{X}^\ell$ *that lie on* $P_{\mathrm{G}'}(v_1, v_2)$ *in* $\mathrm{T}_{\mathrm{G}'}$*. Furthermore, all other classes in* G *are classes in* G'*, and there are no other classes in* G'*.*

By applying Corollary B.2 and Lemma B.1 we get:

Lemma B.7 *Let* C_1 *and* C_2 *be two 3-classes in a 2-connected graph* G *which correspond to leaves in* T_G*. Then there exists an edge* $e_1 = (u_1, v_1)$ *between vertices in* C_1 *and an edge* $e_2 = (u_2, v_2)$ *between vertices in* C_2*, for which the following is true. If we remove* e_1 *and* e_2*, and add an edge between* u_1 *and* u_2*, and an edge between* v_1 *and* v_2 *then the vertices belonging to 3-components and bars residing on the simple path between* C_1 *and* C_2 *in* T_G *form a 3-class in the modified graph* G'*.*

Proof: Since C_1 and C_2 both correspond to 3-component leaves in T_G, they (i.e. the nodes they correspond to) are each connected to a single bar in T_G. Let the bars they are connected to be $\{w_1, x_1\}$ and $\{w_2, x_2\}$, respectively (where $\{w_1, x_1\}$ and $\{w_2, x_2\}$ might coincide). Then in each C_i there exist two vertices u_i, v_i which are neither x_i nor w_i, and which have an edge between them. Let e_i be this edge.

Similarly to Lemma B.3, let $\mathrm{Y}^1, \ldots, \mathrm{Y}^\ell$ be the sequence of bars and 3-components on the path between C_1 and C_2 in T_G (here $\mathrm{Y}^1 = \mathrm{C}_1$, $\mathrm{Y}^\ell = \mathrm{C}_2$, and using the notation above, $\mathrm{Y}^2 = \{w_1, x_1\}$ and $\mathrm{Y}^{\ell-1} = \{w_2, x_2\}$). Consider first the removal of e_1 and let G_1 be the resulting graph. Then by Corollary B.6, in G_1 the vertices belonging to $\mathrm{C}_1 \setminus \{u_1, v_1\}$ are possible divided (in a non-disjoint manner) among 3-components and bars, $\mathrm{X}_1^1, \ldots, \mathrm{X}_1^{\ell_1}$ which lie on $P_\mathrm{G}(u_1, v_1)$. This path either contains the bar $\mathrm{Y}^2 = \{x_1, w_1\}$ or there exists a 3-component X_1^j which contains w_1 and x_1, and it is connected to Y^2 in $\mathrm{T}_{\mathrm{G}_1}$. Furthermore, since C_1 is a leaf in G, there are two possibilities for $\mathrm{S}_{\mathrm{G}_1}(u_1)$ ($\mathrm{S}_{\mathrm{G}_1}(v_1)$). It is either a single node subtree, consisting of a 3-component leaf in $\mathrm{T}_{\mathrm{G}_1}$, or it is a three node subtree, consisting of two edge components (with u_1 (v_1)) being a common edge point and a cycle node they are connected to. In the former case X_1^1 is the abovementioned 3-component, and in the latter, X_1^1 is a bar on the cycle, and the cycle does not include any other edges or bars.

Next, we remove e_2 and obtain a graph G_2. Similarly, in G_2 the vertices in $\mathrm{C}_2 \setminus \{u_1, v_2\}$ are possible divided (in a non-disjoint manner) among components and bars $\mathrm{X}_2^1, \ldots, \mathrm{X}_2^{\ell_2}$ which lie on $P_{\mathrm{G}_2}(u_2, v_2)$. This path either contains the bar $\mathrm{Y}^{\ell-1} = \{x_2, w_2\}$ or there exists a 3-component X_2^j which contains w_2 and x_2, and it is connected to $\mathrm{Y}^{\ell-1}$ in $\mathrm{T}_{\mathrm{G}_2}$. The subtrees $\mathrm{S}_{\mathrm{G}_2}(u_2)$ and $\mathrm{S}_{\mathrm{G}_2}(v_2)$ have one of the two structures defined above for $\mathrm{S}_{\mathrm{G}_1}(u_1)$. By Lemma B.5, if we now add an edge between u_1 and u_2 and and edge between v_1 and v_2 then u_1, v_1, u_2, v_2 together with all vertices in $\mathrm{X}_1^1, \ldots, \mathrm{X}_1^{\ell_1}, \mathrm{Y}^2, \ldots, \mathrm{Y}^\ell, \mathrm{X}_2^1, \ldots, \mathrm{X}_2^{\ell_2}$ become a class in the augmented graph. This set of vertices is exactly the union of C_1, C_2 and the vertices belonging to 3-components and bars on the path between C_1 and C_2 in G.

■

Lemma B.8 *Let* G *be a 2-connected graph whose auxiliary graph,* T_G*, has* L_1 *degree-2 vertices and* L_2 *3-component leaves. Then by removing and adding at most* $4(L_1 + L_2)$ *edges to* G *we can* *transform it into a 3-connected graph* G′. *Furtheremore, suppose that the maximum degree of* G *is* d. *Then the maximum degree of* G′ *is upper bounded by* $\max\{d, 3\}$ *if either* $d > k$ *or* dN *is even and by* 4 *otherwise.*

Proof: We start by noting that degree-2 vertices are vertices which belong to a single edge component or to two edge components that lie on a cycle. These vertices do not belong to bars. Assume first that L_1 is even. In such a case we pair the degree-2 edges (not allowing pairs that are already end-points of a common edge), and add an edge between vertices in each pair. The degree of these vertices is now 3. By Lemma B.5, these pairs of vertices (and possibly others) become 3-connected, and in the resulting graph, G′, each vertex belongs to some 3-components and/or to some bar in $T_{G'}$. In particular, each vertex either belongs to a 3-component leaf or is on the path of 3-components and bars between two such leaves. Furthermore, the number of 3-component leaves is at most $L_1/2 + L_2$. Applying Lemma B.7 at most $L_1/2 + L_2$ times, we can merge all classes of G′ into a single 3-class by removing and adding at most $4(L_1/2 + L_2)$ edges in and between the 3-component leaves of $T_{G'}$. Note that in the process described in Lemma B.7 we do not increase the degree of any vertex.

In case L_1 is odd, we can pair all but one degree-2 vertex, v. We separate in to two subcases: N is even, and N is odd. In case N is even, simple counting shows that it cannot be the case that all vertices but v have degree exactly 3, and thus the maximum degree of G is at least 4. In case N is odd then by the Corollary statement the resulting graph is allowed to have degree 4 vertices. If we have at least one vertex with degree smaller than the maximum between the largest degree in G and 4, then we add an edge between this vertex and v. Otherwise (all vertices except v have degree at least 4), it is not hard to verify that by removing an arbitrary edge (u_1, u_2) and adding the edges (v, u_1) and (v, u_2), the graph becomes 3-connected. ■

C Proof of one inequality

Our aim is to prove that for any integers $c \le D$,

$$p \stackrel{\text{def}}{=} \prod_{i=0}^{n-3} \left(\frac{n - i - (2c/D)}{n - i} \right) > (2n)^{-2c/D}$$

A proof that $p = \Omega(n^{-2c/D})$, for constant c, D, can be found in Karger's Ph.D. Thesis [Kar95] (see proof of Corollary 4.7.5 which refers to an exercise in Knuth Vol. 1). An alternative proof follows. Fixing D, c and n, let $f(m) \stackrel{\text{def}}{=} \prod_{i=0}^{n-3} \frac{m-Di-2c}{m-Di}$. Then, for any $m > D(n-3)+1$, we have $f(m) > f(m-1)$. In particular, for every $j \le D-1$, we have $f(Dn) > f(DN-j)$, and so

$$\begin{aligned} p^D &= f(Dn)^D \\ &> \prod_{j=0}^{D-1} f_{D,n,c}(Dn-j) \\ &= \prod_{j=0}^{D-1} \prod_{i=0}^{n-3} \frac{Dn - j - Di - 2c}{Dn - j - Di} \\ &= \prod_{k=0}^{D \cdot (n-3)+(D-1)} \frac{Dn - k - 2c}{Dn - k} \\ &= \prod_{\ell=0}^{2c-1} \frac{2D - \ell}{Dn - \ell} \end{aligned}$$

where the third equality is obtained by substituting $k = Di + j$. Finally, we get

$$\begin{aligned} p &> \left(\frac{(2D/4)^{2c}}{(Dn)^{2c}}\right)^{1/D} \\ &= (2n)^{-2c/D} \end{aligned}$$

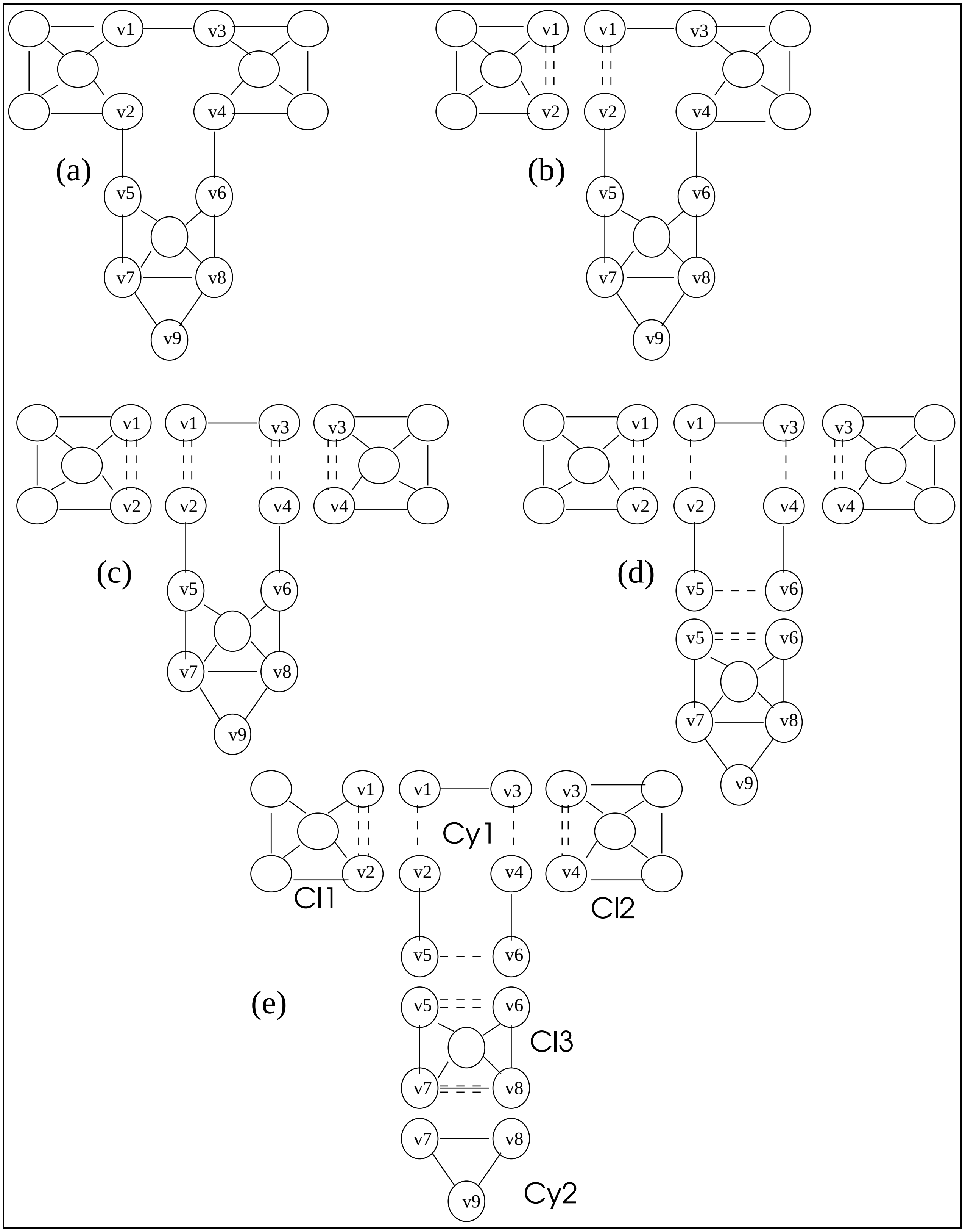

Figure 1: (a): A 2-vertex-connected graph. (b)–(e): The graph's recursive decomposition into 3-components and cycles, using the separating pairs (v_1, v_2), (v_3, v_4), (v_5, v_6), and (v_7, v_8), respectively. The edges added in the decomposition process are depicted as dotted lines (in all cycles, multiple edges are merged into single edges).

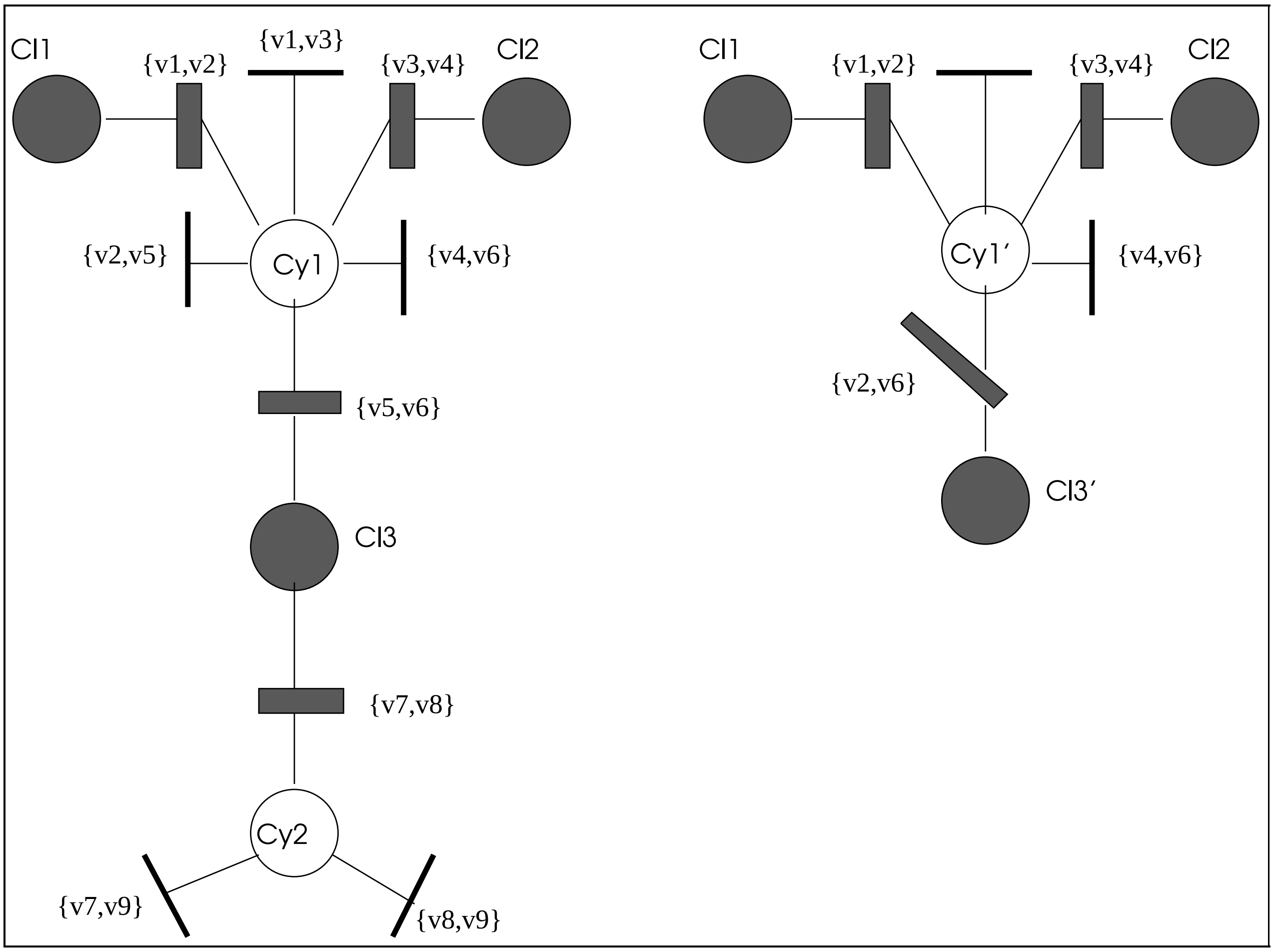

Figure 2: Left: The auxiliary tree of the graph depicted in Figure 1. 3-components are denoted by lightly filled circles, cycles by non-filled circles, blocks by lightly filled rectangles, and edge components by bold lines. Right: The auxiliary tree of the same graph with an edge added between v_2 and v_9. The new component Cl_3' is the union of v_2, v_9, and the vertices v_5, v_6, v_7, v_8 which reside in the components and blocks on the path between the subtrees corresponding to v_2 and v_9.