Property Testing in Bounded Degree Graphs

Oded Goldreich* Dana Ron†

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 $\epsilon \cdot (dN)$ 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.

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 [7, 21, 13, 4, 3, 11, 1] and codes [3, 1, 5, 16], and its relevance to the construction of probabilistically checkable proofs [4, 3, 11, 2, 1] is well known. Recently, a general formulation of property testing has been presented in [14], 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 [14] 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, Goldwasser and Ron, as part of a general study of *Property Testing* [14]. 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 [14] 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 every N-vertex graph is $\frac{2}{N}$-close to being connected and so the algorithm may as well accept any graph).

*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] 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).

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 [14] do not apply to our model and the analogies of most of the results in [14] 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 $k = 2,3$)**:** 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 [14]. Recall that in [14] 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 [14] works in time $\text{poly}(1/\epsilon)$ 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 [1, 20], 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 [22, 19, 12, 6, 18].

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 [17]. For $k = 2$ (resp. $k = 3$), we have an alternative algorithm which works in time linear (resp. quadratic) in the size of the component. 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 $k \geq 1$). Testing cycle-freeness is considered in Section 4, and our hardness results are presented in Section 5. Our algorithms for testing k-vertex-connectivity (for $k = 2, 3$) and other testing algorithms (e.g., for Planarity) as well as further details concerning claims presented in this extended abstract can be found in the full version of this paper [15].

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 V(G) its vertex set and by E(G) its edge set. We assume, without loss of generality, that $\mathrm{V(G)} = [|\mathrm{V(G)}|] \stackrel{\text{def}}{=} \{1, \ldots, |\mathrm{V(G)}|\}$ and that for every vertex $v \in \mathrm{V(G)}$, 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, $(u, v) \in \mathrm{E(G)}$ 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 $f_{\mathrm{G}} : \mathrm{V(G)} \times [d] \mapsto \mathrm{V(G)} \cup \{0\}$, 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 $\mathrm{V(G_1)} = \mathrm{V(G_2)} = [N]$, is defined as follows:

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

This notation is extended naturally to a set, $\mathcal{C}$, of N-vertex graphs with degree bound d; that is, $\mathrm{dist}(\mathrm{G}, \mathcal{C}) \stackrel{\text{def}}{=} \min_{\mathrm{G}' \in \mathcal{C}} \{\mathrm{dist}_d(\mathrm{G}, \mathrm{G}')\}$. 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 $\mathcal{A}$.

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 $1 - \delta$.[3] 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 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 $\mathrm{poly}(\frac{k}{\epsilon})$. In particular,*

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

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

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.*

The lemma is very easy to establish when the sum of degrees in each connected component C_i is at most $d \cdot |\mathrm{C}_i| - 2$. In this case we simply connect the components by adding edges between vertices whose degree is less than d. Otherwise we must first remove edges so as to obtain such "non-saturated" vertices while taking care not to disconnect any component.

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.

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)[4] starting from s until $\frac{8}{\epsilon d}$ 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 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.

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

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 graph 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 $\mathrm{X} \subseteq \mathrm{V}$ is said to be k–edge-connected if there are k edge-disjoint paths between each pair of vertices in X. (Any singleton is defined to be k–edge-connected.) We stress that, in case $k \geq 3$, these paths may go through vertices not in X. 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. 3.2.5) remove this assumption. We next state some facts, necessary for our algorithms, concerning the structure of $(k-1)$–connected graphs. Let G be a $(k-1)$–connected graph. Then we can define an auxiliary graph T_G [9] (based on the *cactus* structure of [8]), which is a tree, such that for every k-class in G there is a corresponding (unique) node in T_G. The tree 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 T_G consists of a single node, corresponding to the vertex set of G. Otherwise, T_G has at least two leaves. The leaves of 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 $k-1$. The analysis of our algorithm relies on the following lemma, which directly follows from Lemma A.4 in 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* $k \cdot |V(G)|$ *is even.*[5] *Then,* T_G *has at least* $\frac{\epsilon}{8}dN$ *leaves.*

3.2.2 The Algorithm

The above lemma implies that at least $\frac{\epsilon dN}{16}$ of the leaves in 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.

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.

This algorithm can be modified analogously to the improved version of the connectivity tester, yielding a saving of a $\tilde{O}(1/\epsilon d)$ factor in the running-time.

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 two subsections, we present such (k-class leaf) identification algorithms for the cases $k = 2, 3$ 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 k-class Leaf (for $k = 2, 3$)

We start with the case $k = 2$. 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 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 T_1 be the tree defined by the search, and let $E(T_1)$ 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 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 X_2 be the set of vertices reached.
3. If there is a single edge with one end-point in X_2 and the other outside of X_2 (i.e. $(X_2, \overline{X_2})$[6] is a cut of size 1), then 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* T_G*, and* s *a vertex in* C*. Then the 2-Class Leaf Identification procedure will always terminate with* $X_2 = C$*.*

Proof: Since the first DFS terminates after seeing $n+1$ vertices, and vertices in $\overline{C}$ can be reached only by traversing the single edge (u, v) where $u \in C$ and $v \in \overline{C}$, 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, $X_2 \subseteq C$. What needs to be shown is that the second search reaches *every* vertex in C (i.e., $X_2 = C$), and hence the cut $(C, \overline{C})$ is discovered.

Assume contrary to this claim, that $S \stackrel{\text{def}}{=} C \setminus X_2$ is non-empty. Let $(u_1, v_1), \ldots, (u_\ell, v_\ell)$ be the set of edges crossing the cut (X_2, S), where $\forall i$, $u_i \in X_2$ and $v_i \in S$. Since C is 2–connected, there must be at least two edges in the cut (X_2, 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 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. ■

Identifying a 3-class Leaf of a 2-Connected Graph. 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 2-Class Leaf Identification procedure (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*

[5] The reason for this technical requirement is to rule out the pathological case in which $d(=k)$ and $|V(G)|$ are both odd in which case the class of k–connected graphs with N vertices and max-degree d is empty. Clearly, this pathological case is easily detected by the algorithm.

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

in C*. Then the above 3-Class Leaf Identification procedure terminates in finding the cut* $(\mathrm{C}, \overline{\mathrm{C}})$.

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

3.2.4 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). Our algorithm for finding k-class leaves is based on Karger's Contraction Algorithm [17] 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 $(\mathrm{S}, \overline{\mathrm{S}})$ of size less than k is found: Starting from the singleton set $\{s\}$, at each step the algorithm maintains a set S of vertices it has visited. As long as $|\mathrm{S}| < n$ and the cut $(\mathrm{S}, \overline{\mathrm{S}})$ has size at least k, the algorithm chooses an edge to traverse among the cut edges in $(\mathrm{S}, \overline{\mathrm{S}})$ 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 *a leaf class 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* $(\mathrm{C}, \overline{\mathrm{C}})$.

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) [17]. 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 [17]. 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 that 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. In our case $c = k-1$ and, since all cuts in G' other than the minimum cut $(\mathrm{C}, \{x\})$ have size at least k, we can set $D = k$. The lemma follows. ■

3.2.5 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 leaves 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 $d \geq k+1$.*[7] *Then the minimum 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, 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.

   (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 i-class leaves of certain small sizes. Specifically, an i-class leaf C is hit with probability $\frac{|\mathrm{C}|}{N}$ and is identified as such (with high probability) within time $\mathrm{T}_i(|\mathrm{C}|)$ (see Sec. 3.2.2). Note that C is not necessarily an i-class leaf 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

[7] 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.

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. 3.2.4 to C′. It follows that if a vertex $s \in \mathrm{C}'$ is chosen by the (modified) algorithm in iteration $\ell = \lceil \log(|\mathrm{C}'|) \rceil$ (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 $(\mathrm{C}', \overline{\mathrm{C}'})$, with high probability, within time $\mathrm{T}_j(2 \cdot |\mathrm{C}'|)$. The above analysis holds also with respect to the (modified) identification procedures for 2-class and 3-class leaves.

4 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 $\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_L$ 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 $N - L$. Intuitively, if G is far from being cycle-free, then this is due either to many small components having at least one extra edge each, or to many extra edges within the (few) big components. In the first case, we can hope to sample a bad small component. In the second case, we may consider the subgraph of G which consists of all big component and detect a discrepancy between its edge count and its vertex count. 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 $n_i \stackrel{\mathrm{def}}{=} |\mathrm{C}_i|$, and let $b_i \stackrel{\mathrm{def}}{=} m_i - (n_i - 1) \geq 0$ 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 $b \stackrel{\mathrm{def}}{=} \sum_{i=1}^{L} b_i$ 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 connected 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). If G is cycle-free the we should have $\sum_{i=1}^{t} m_i = \sum_{i=1}^{t} (n_i - 1); = (\sum_{i=1}^{t} n_i) - t$. Therefore 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.

This leads to the following (for details see [15]).

Theorem 4.1 *There exists a testing algorithm for the Cycle-Free property whose query complexity and running time are* $O(\frac{1}{\epsilon^3 d})$.

REMARK: The tester referred to in the theorem 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 5.1: 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 seen. 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).

5 Hardness Results

In this section we present lower bounds on the query complexity of algorithms for testing the Bipartite and Expander properties.

5.1 Testing Bipartiteness

A graph is said to be *bipartite* if its vertices can be partitioned into two sets so that there are no edges with both endpoints in the same set.

Theorem 5.1 *Testing Bipartiteness with distance parameter 0.01 requires* $\frac{1}{3} \cdot \sqrt{N}$ *queries.*

Proof: For any even[8] 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 a 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. Using a probabilistic argument, we can show that almost all graphs in $\mathcal{G}_1^N$ are far from being bipartite. However, we also show that a testing algorithm which performs $o(\sqrt{N})$ queries is not likely 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). The intuition is that by performing less than $\sqrt{N}$ queries, the tester is not likely to observe a cycle. Therefore, it will not be able to distinguish between the case in which the graph belongs to $\mathcal{G}_2^N$ (and has only cycles of even length) and the case in which the graph belongs to $\mathcal{G}_1^N$ (and has "many" cycles of odd length). More precisely,

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

we describe two probabilistic processes which answer a testing algorithm's queries while constructing a random graph in $\mathcal{G}_1^N$ or $\mathcal{G}_2^N$, respectively. We then show that for every given testing algorithm which performs $\alpha\sqrt{N}$ queries, the statistical difference between the distributions induced by the two processes on query-answer sequences, is small (i.e., $4\alpha^2$). For details see [15].

5.2 Testing Whether a Graph is an Expander

The *neighbor set* of a set A of vertices of a graph $\mathrm{G} = (V, E)$, denoted $\Gamma(A)$, is defined as follows: $\Gamma(A) \stackrel{\text{def}}{=} A \cup \{u : (v,u) \in \mathrm{E}, v \in A\}$. A graph on N vertices is an (N, α, β)-expander if for every subset A of the vertices which has size at most αN, $|\Gamma(A)| \geq \beta|A|$. Let us set $\alpha = \frac{1}{4}$ and $\beta = \frac{6}{5}$, and simply refer to an $(N, \frac{1}{4}, \frac{6}{5})$-expander, as an expander. Here we show that

Theorem 5.2 *Testing whether a graph is an expander, with distance parameter 0.01, requires* $\frac{1}{5} \cdot \sqrt{N}$ *queries.*

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 details see [15].

Acknowledgments

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

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and intractability of approximation problems. In *33rd FOCS*, pages 14–23, 1992.

[2] S. Arora and S. Safra. Probabilistic checkable proofs: A new characterization of NP. In *33rd FOCS*, pages 1–13, 1992.

[3] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *23nd STOC*, pages 21–31, 1991.

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

[5] M. Bellare, O. Goldreich, and M. Sudan. Free bits, pcps and non-approximability – towards tight results. In *36th FOCS*, pages 422–431, 1995. Full version available from *ECCC*, `http://www.eccc.uni-trier.de/eccc/`.

[6] A. Benczur. A representation of cuts within 6/5 times the edge connectivity with applications. In *36th FOCS*, pages 92–101, 1995.

[7] 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.

[8] 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.

[9] 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.

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

[11] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *32nd FOCS*, pages 2–12, 1991.

[12] H. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *32nd FOCS*, pages 812–821, 1991.

[13] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *23nd STOC*, pages 32–42, 1991.

[14] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. In *37th FOCS*, pages 339–348, 1996.

[15] O. Goldreich and D. Ron. Property testing in bounded degree graphs. Available from `http://theory.lcs.mit.edu/~danar`, 1997.

[16] J. Håstad. Testing of the long code and hardness for clique. In *28th STOC*, pages 11–19, 1996.

[17] D. Karger. Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm. In *4th SODA*, pages 21–30, 1993.

[18] H. Nagamochi and T. Ibaraki. Deterministic $\tilde{O}(nm)$ time edge-splitting in undirected graphs. In *28th STOC*, pages 64–73, 1996.

[19] D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge-connectivity. In *31st FOCS*, pages 698–707, 1990.

[20] C.H. Papadimitriou and M. Yanakakis. Optimization, approximation and complexity classes. *JCSS*, 43:425–440, 1991.

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

[22] T. Watanabe and A. Nakamura. Edge-connectivity augmentation problems. *JCSS*, 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., [10]), 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 [9] for more details. We stress that the graphs below are not necessarilty simple; that is, parallel edges are allowed.

Fact A.1 (cf., [9]): *Let $k > 1$ be an integer and G be a $(k-1)$-connected graph. Then there exists an auxiliary graph, $\mathrm{T_G}$, 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.*

The reader can easily verify the above facts for the well known case of $k = 2$ (where all tree nodes correspond to 2-connected classes).

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 C_i and C_j be two nodes on the cycle Cy. 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: If C_i and C_j were adjacent on the cycle then the cycle remains the same except that C_i and C_j are replaced by C_k. If C_i and C_j were not adjacent then the cycle Cy is split into two cycles, where C_k belongs to both and the other nodes are partitioned according to their position on the cycle (with respect to C_i and C_j) in the obvious manner. In either case, if as the result of the squeezing we get a cycle of size 2, then this cycle is replaced by a single tree edge between the two nodes on the cycle.

The following lemma is well known in case $k = 2$.

Lemma A.2 (cf., [9]): *Let* G *be a* $(k-1)$*-connected graph, and* $\mathrm{T_G}$ *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* $\mathrm{T_G}$ *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 [8]. In [15] we provide an alternative (direct) proof. 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,* $\mathrm{T_G}$ *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* $\mathrm{T_G}$ *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.*

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

Lemma A.4 *Let* G *be a* $(k-1)$*-connected graph, whose auxiliary graph,* $\mathrm{T_G}$*, has* L *leaves. Then by removing and adding at most* $4L$ *edges to* G *we can transform it into a* k*-connected graph* G'*. Furthermore, 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 [19]. 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 $\mathrm{T_G}$ 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 $d' \stackrel{\text{def}}{=} \max\{d, k\}$. 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 (between the singleton and some other vertex) and increase the degree of the resulting graph to $d' + 1 = k + 1$. ■