Connectivity Testing in the Bounded-Degree Model: Algorithms, Implementation, and Analysis

Paul Sigloch
Vancouver Island University

December 5, 2025

Abstract

This study examines different approaches for testing connectivity in bounded-degree graphs. Therefore, we first introduce the bounded-degree model and present a basic approach for testing connectivity that runs in linear time. Then, we delve into the concept of sublinear testing, where we employ a technique that avoids visiting every vertex of a graph in exchange for a small margin of error. Based on this idea, the paper presents two efficient algorithms, where the second algorithm improves the search strategy of the first one. We formally proved the correctness of both algorithms and analyzed their efficiency based on the number of queries they make to a given input graph. Additionally, both testers were implemented in Python and evaluated empirically to confirm the theoretical bounds and to study practical constant factors. Our comparison showed that the second algorithm becomes more efficient compared to the first one as the acceptable error rate decreases. Overall, this work contributes two efficient connectivity testing algorithms and establishes a theoretical foundation applicable to other property testing concerns.

1 Introduction

Graphs have become essential tools in various applications, transcending the domains of mathematics and theoretical computer science. Understanding the connections within graph networks is crucial in diverse applications such as social media networks, communication networks, and biological pathways [7, 9]. In protein-protein interaction networks, for instance, connectivity reveals how proteins interact, ultimately influencing cellular processes [11]. Moreover, the exponential growth of data in today's information age has led to increasingly complex graphs within these applications, often containing millions of vertices. At the same time, modern applications demand real-time processing and fast response times, while aiming to minimize the consumption of computational resources for each query. As the number of vertices increases, traditional connectivity testing algorithms become resource-intensive, affecting performance and usability. In this context, the search for sublinear or even constant-time algorithms becomes crucial for practical applications [12].

This paper addresses the design of such algorithms and explores techniques necessary for building efficient connectivity testers. In doing so, we closely follow the testing approaches proposed by Bhattacharyya & Yoshida [1] and Goldreich & Ron [4], building on their theoretical foundations and algorithmic constructions. We begin in Section 2 with essential preliminaries on graph theory, covering topics such as the bound-degree model and the fundamentals of sublinear connectivity testing. In Section 3, we introduce the sublinear connectivity testing techniques that form the basis of the algorithms presented in Section 4. Section 5 then focuses on the analysis of the runtime and query complexity of each algorithm, followed by a comparative evaluation of their efficiency. Finally, Section 6 presents an implementation-based study of both testers, highlighting practical considerations and empirical behaviour that complement the theoretical analysis.

2 Model and Theory

2.1 Preliminaries

We consider graphs of the form $G = (V, E)$, where V is the set of vertices and E is the set of edges, with the total number of vertices denoted by $n := |V(G)|$. Furthermore, all graphs are finite, undirected, and have no multiple edges between two vertices or self-loops on the same vertex. Hereby, the degree of a vertex v is denoted as $deg(v)$, indicating the number of edges incident to the vertex. A sequence of vertices is called a *path* if each consecutive pair of involved vertices is connected by an edge. Specifically, we refer to a path as a u-v *path* if it connects two vertices u and v belonging to the same vertex set. If a u-v path can be established between any two vertices within a graph G, we call G fully connected. Moreover, we define a *connected component* in a graph G as a maximal subset of vertices where every pair of vertices is connected by a path. Consequentially, a fully connected graph G has exactly one connected component. If G has more than one connected component, we call G not connected [1, 9].

2.2 Bounded-Degree Model

This subsection introduces the bounded-degree model, with definitions and notation based on the work of Bhattacharyya & Yoshida [1] and Goldreich & Ron [4].

Let G be a graph belonging to the class of bounded-degree graphs. Then every vertex $v \in G$ has a degree that is constrained by a positive fixed integer $d \in \mathbb{N}_{>0}$. Furthermore, we can represent G as a function:

$$f_G : V(G) \times [d] \to V(G) \cup \{0\}.$$

Hereby, $f_G(v, i) = u$ is true if vertex u is the i^{th} neighbor of v, while $f_G(v, i) = 0$ is true if v has less than i neighbors. To show the distance $dist_d(G, G')$ between two graphs G and G' with the same set of vertices $[n]$ and the same degree bound d, we can express this as

$$\frac{|(v,i) : v \in [n], i \in [d] \ \wedge \ f_G(v,i) \neq f_{G'}(v,i)|}{d \cdot n},$$

which, in other words, is the proportion of differing vertices in terms of their edge connections between the two graphs. In addition, query access to G and its vertices is granted. This allows checking certain basic properties of G, such as the total number of vertices, the degree of each vertex in V, and the existence of an edge between any two vertices u and v of V.

2.3 Basic Connectivity Testing

Before we look at more sophisticated testing approaches, we first examine how we actually conduct connectivity testing in graphs. Therefore, we utilize a graph traversal algorithm such as the breadth-first search (BFS) on an input graph G. In doing so, we start by randomly selecting a vertex $v \in V(G)$ as the root node and explore all of its neighboring vertices on the same level. Each traversed vertex is marked as visited, and its respective neighbors are then explored in the same way, level by level. This process continues until all vertices of $V(G)$ have been visited, or until no more unvisited vertices can be found. By checking whether all vertices have been traversed, we can determine the connectivity of the graph [8].

Although this approach has a time complexity of $\mathcal{O}(n)$, it does not scale well for input graphs with large n. Thus, there is need for a more efficient method that can traverse even large graphs in sublinear time, or ideally in constant time [12].

2.4 The ε-Parameter

To achieve sublinear time complexity when testing a graph G with n vertices for connectivity, it is required that not all vertices can be visited. Therefore, we want to determine by approximation whether G is connected, while accepting a small margin of error. To accomplish this, we introduce the constant ϵ, where $\epsilon \in (0, 1]$. Building on the formal definition of the bounded-degree model from Subsection 2.2 and similar to the prior work of Bhattacharyya & Yoshida [1], we now define ϵ-closeness and ϵ-farness.

Definition 1. *Let G be a graph with degree bound $d \in \mathbb{N}_{>0}$. We say that G is **ε-close** to a property $\mathcal{P}$ if the number of edge modifications required to transform G into a graph $G' \in \mathcal{P}$ is less than $\epsilon \cdot d \cdot n$. Edge modifications involve the removal or addition of edges to the graph G. Conversely, we say that G is **ε-far** from $\mathcal{P}$ if G is not ϵ-close.*

Here, property $\mathcal{P}$ from Definition 1 refers specifically to the connectivity of the graph. This means that we only consider adding edges when speaking of modifying edges. Additionally, we prefer a small value for ϵ within the range $(0, 1]$ since this requires the graph to be closer to being connected before we accept it, making the test more stringent.

3 Sublinear Testing for Connectivity

3.1 Baseline for a First Simple Approach

Introducing the concept of ϵ-farness allows us to derive a statement about the number and size of connected components within a graph G that is ϵ-far from being connected. The computation of these properties helps us to derive optimal termination criteria for connectivity testing algorithms in the form of constraints on iterations and visited vertices per BFS. We start by considering the minimum number of contained connected components that must be present within a graph G that is ϵ-far from being connected. This leads to the following theorem, based on Bhattacharyya & Yoshida [1].

Theorem 3.1. *If a graph G is ϵ-far from being connected, then the number of connected components in G is greater than $\epsilon \cdot d \cdot n$.*

Proof. Let G be a graph with at most $\epsilon \cdot d \cdot n$ connected components. By adding $\epsilon \cdot d \cdot n - 1$ edges to the unconnected components, we can derive a fully connected graph. Then, graph G would no longer be ϵ-far from being connected. Note that we used less than $\epsilon \cdot d \cdot n$ edges, which falls below the threshold for G to be ϵ-far from connected, as covered in Definition 1. Hence, we reach a contradiction. QED

While Theorem 3.1 helps us determine how many connected components must at least be contained in a graph G that is ϵ far from being connected, we also want to make an approximation about the size of some of those connected components. Deriving a threshold for some connected component sizes will eventually help us to terminate a BFS early, since we do not want to traverse the whole graph if G is fully connected. For this reason, we compose the following corollary, which is also based on Bhattacharyya & Yoshida [1].

Corollary 3.1.1. *If a graph G is ϵ-far from being connected, then the number of connected components of size less than $\frac{2}{\epsilon \cdot d}$ vertices in G is at least $\frac{\epsilon \cdot d \cdot n}{2}$.*

Proof. Towards a contradiction, we assume that a graph G contains at least $\frac{\epsilon \cdot d \cdot n}{2}$ connected components with size greater than or equal to $\frac{2}{\epsilon \cdot d}$ vertices. In one case, this means that we have more than

$$\frac{\epsilon \cdot d \cdot n}{2} \cdot \frac{2}{\epsilon \cdot d} = n$$

vertices in G. Since G contains exactly n vertices, that is a contradiction. In another case, this implies that we have exactly $\frac{\epsilon \cdot d \cdot n}{2}$ connected components inside G, each with size of $\frac{2}{\epsilon \cdot d}$ vertices. From the proof of Theorem 3.1 we know that G must have more than $\epsilon \cdot d \cdot n$ connected components, otherwise we can make G connected by adding $\epsilon \cdot d \cdot n - 1$ edges. Thus, if G has exactly $\frac{\epsilon \cdot d \cdot n}{2}$

connected components of size $\frac{2}{\epsilon \cdot d}$, it can not have any more connected components that would classify G as ϵ-far from being connected. Once again, we reach a contradiction.

QED

It is important to note that we are merely setting a lower bound on the number of connected components of a certain size. Still, we must consider that we may contain connected components with size greater than $\frac{2}{\epsilon \cdot d}$. Furthermore, we call components *small* if they contain less than $\frac{2}{\epsilon \cdot d}$ vertices, and classify them as *large* otherwise [1].

Remark 1. *Note that within the expression $\frac{\epsilon \cdot d \cdot n}{2}$ of Corollary 3.1.1, any alternative positive fixed integer x may be chosen as the denominator instead of 2. Thereby, it must be ensured that the resulting value and the corresponding threshold for the connected component size of less than $\frac{x}{\epsilon \cdot d}$ vertices remain a positive number greater than or equal to 2. In addition, Theorem 3.1 must still hold for these values.*

Another observation that can be drawn from Definition 1, which restricts the possible values for ϵ in combination with d when considering ϵ-farness, is presented in the following lemma, derived from [1].

Lemma 3.2. *If a graph G is ϵ-far from being connected, then $\epsilon \cdot d < 1$.*

Proof. Let G be a graph that is ϵ-far from being connected. Towards a contradiction, we assume that $\epsilon \cdot d \geq 1$. By Definition 1, a graph is ϵ-close to being connected, if it can be made connected by adding less than $\epsilon \cdot d \cdot n$ edges. Thereby, we add $n - 1$ edges, which makes G connected, since every graph can be made connected by adding $n - 1$ edges. Hence, we reach a contradiction.

QED

From here, we have everything we need to construct a first simple connectivity tester using the statements of Theorem 3.1 and Corollary 3.1.1. This algorithm is shown and explained in Subsection 4.2. However, there is an additional improvement that we can make by examining Corollary 3.1.1 more closely, which is presented below.

3.2 Groundwork for Improved Connectivity Testing

So far, we have only classified components into smaller and larger components. However, the vertices within these components are not necessarily evenly distributed across all possible sizes. Instead, there is a statistically expected skew toward smaller components. This observation can also be derived by following the proposition of Remark 1, which concerns the substitution of values other than 2 in the expressions of Corollary 3.1.1. The skewness implies that there are likely to be significantly more components with fewer vertices than those with more vertices. This inherent tendency is especially noticeable under the constraints of bounded-degree graphs, where isolated vertices and pairs of connected vertices are more likely to be found than larger clusters [6].

To exploit this skewed distribution and improve our search efficiency, we categorize small connected components based on their size ranges. We achieve this by wrapping a logarithmic function around the threshold for the size of small components, which we obtain from Corollary 3.1.1. This function is likewise based on Bhattacharyya & Yoshida [1]:

$$\ell = \lceil \log_2 \left(\tfrac{2}{\epsilon \cdot d} + 1 \right) \rceil. \tag{1}$$

Here, (1) defines the number of sets of connected components sizes and provides the underlying structure for categorization. Furthermore, we use C_i to denote a set of connected components which contains at least 2^{i-1} but less than 2^i vertices, where $i \in [\ell]$. This allows us to generate disjoint intervals for the set sizes of all small connected components, where we ensure at the same time that all these small connected component sizes are upper bounded by $2^\ell - 1$ vertices. Importantly, the size range defined by the logarithm is exponential. As the set index increases, the size range doubles, allowing for a fine-grained analysis of the distribution. This approach leverages the existing skew towards smaller components by focusing our search on the more probable smaller size ranges.

Note that we add 1 to the result of $\frac{2}{\epsilon \cdot d}$ for the case that $\epsilon \cdot d$ produces a value greater than 2. We also use a ceiling function on the logarithmic wrapper function to get only positive integers as result. From this position we arrive at the following theorem, based on Bhattacharyya & Yoshida [1], which is a further refinement of Corollary 3.1.1 regarding the inclusion of the component size distribution along with (1).

Theorem 3.3. *If a graph G is ϵ-far from being connected, then there exists $i \in [\ell]$ such that $|C_i| \geq \frac{\epsilon \cdot d \cdot n}{2 \cdot \ell}$.*

Proof. Let G be a graph that is ϵ-far from being connected. Towards a contradiction, we assume that for all $i \in [\ell]$ is given that

$$|C_i| < \tfrac{\epsilon \cdot d \cdot n}{2 \cdot \ell}.$$

This implies that

$$\sum_{i=1}^{\ell} |C_i| < \tfrac{\epsilon \cdot d \cdot n}{2}.$$

But from Corollary 3.1.1 we know that the number of connected components of size less than $\frac{2}{\epsilon \cdot d}$ must be at least $\frac{\epsilon \cdot d \cdot n}{2}$. Hence, we reach a contradiction.

QED

Theorem 3.3 allows us to build another algorithm that is more fine-tuned with respect to the approximate distribution of different connected component sizes. This algorithm is presented and examined in Subsection 4.3.

4 Algorithms for Testing Connectivity

4.1 Preliminaries for the Algorithms

In the following, two algorithms based on Subsections 3.1 and 3.2 are presented. Both algorithms are considered to be connectivity testers for a given $\epsilon \in (0, 1]$, and fixed $d \in \mathbb{N}_{>0}$. Moreover, they are both declared to be one-sided error testers, i.e. they accept any connected input graph and reject with a probability of $2/3$ if the input graph is ϵ-far from being connected, which will be shown in their respective proofs of correctness [1]. Thus, both algorithms test graph connectivity for false positives, with no possibility of false negatives. In addition, the vertices are sampled uniformly at random, which is a crucial process for mitigating bias in the analysis of graph properties [3].

As input, both algorithms take a graph G, whereby query access is granted to the graph and its $n := |V(G)|$ vertices. Furthermore, the output type of both algorithms is identical. If the respective algorithm finds evidence that the input graph is ϵ-far from being connected, the boolean value true is returned, otherwise false.

4.2 Simple Tester for Connectivity

Together with Theorem 3.1 and Corollary 3.1.1, we design a simple tester for connectivity as follows in Algorithm 4.1, based on Bhattacharyya & Yoshida [1].

Algorithm 4.1 Simple Connectivity Tester

Input: Graph G

Output: Boolean

1: **for** $t := \Theta(\frac{1}{\epsilon \cdot d})$ **do**
2: Select a vertex $v \in V$ uniformly at random.
3: Conduct a BFS from v on, stop when reaching $\frac{2}{\epsilon \cdot d}$ vertices or no new vertices can be reached.
4: **if** small connected component found (of size $< n$) **then**
5: **return** false.
6: **end if**
7: **end for**
8: **return** true.

In principle, Algorithm 4.1 consists of a single comprehensive loop. If no small connected component is found during t iterations, the boolean value true is returned in line 8, i.e. the input graph G is accepted. At the beginning of each iteration in line 2, a vertex v is sampled uniformly at random. Then a BFS is conducted, where the termination condition is derived from Corollary 3.1.1, which says that the BFS should be terminated if $\frac{2}{\epsilon \cdot d}$ vertices were discovered or if no more new vertices can be found. This can be achieved, for example, by maintaining a list of visited and unvisited vertices, the latter being collected when querying the vertices for their respective

neighbors, whereby duplicate vertices are removed. If the sum of unvisited and visited vertices is equal to or greater than $\frac{2}{\epsilon \cdot d}$ vertices, the current component is said to be large. Otherwise, a small connected component has been found, which means that G can be safely rejected as long as the current component has a size less than n, and thus the boolean value false is returned in line 5.

4.2.1 Proof of Correctness

To prove the correctness of Algorithm 4.1 in terms of being a one-sided error tester for connectivity, we come up with the following lemma.

Lemma 4.1. *Algorithm 4.1 accepts any connected input graph G and rejects with probability at least $2/3$ if G is ϵ-far from being connected. Thus, Algorithm 4.1 is a one-sided error tester for connectivity.*

Proof. Algorithm 4.1 always returns true if an input graph G is connected, since no small connected components can be found within a connected graph. Thus, it accepts any connected input graph.

In addition, we must prove that the input graph G is rejected with probability at least $2/3$ if G is ϵ-far from connected. To achieve this probability, we have to ensure that there are sufficient iterations t, which depends on the hidden constant c. This constant is suppressed in the Theta term in line 1 due to asymptotic simplification. For this reason, in the following probability calculations, we refer to the total number of iterations as

$$s \coloneqq \frac{1}{\epsilon \cdot d} \cdot c, \tag{2}$$

so that c is not suppressed as in the right-hand side of t.

By Corollary 3.1.1, we know that there are more than $\frac{\epsilon \cdot d \cdot n}{2}$ small connected components with size less than $\frac{2}{\epsilon \cdot d}$ inside a graph G that is ϵ-far from being connected. Therefore, the probability that we sample a vertex v contained in a small connected component within one iteration is at least

$$\frac{\frac{\epsilon \cdot d \cdot n}{2}}{n} = \frac{\epsilon \cdot d}{2}. \tag{3}$$

From here we derive the complementary probability of (3), i.e., not sampling a vertex contained in a small component within one iteration, which is less than

$$1 - \frac{\epsilon \cdot d}{2}. \tag{4}$$

Now, together with (4) we calculate the probability of not sampling a vertex contained in a small

component during all iterations, which is less than

$$\left(1 - \frac{\epsilon \cdot d}{2}\right)^s \leq \frac{1}{3}. \tag{5}$$

Here, (5) in turn is needed to derive the probability of not sampling a vertex contained in a small component during all iterations, which is at least

$$1 - \left(1 - \frac{\epsilon \cdot d}{2}\right)^s \geq \frac{2}{3}.$$

Hence, the choice of a sufficiently large hidden constant c is the key to achieving the desired probability of at least $2/3$. Finally, we can state that Algorithm 4.1 is a one-sided error tester for connectivity.

QED

At this point we notice that the hidden constant c from the derivation in the proof of Lemma 4.1 plays a certain role within the success and error probabilities. To find out how strong its dependence on these probabilities is, and also how high we need to set c, at least in a practical scenario, an exemplary calculation of c is shown below.

4.2.2 Calculating the Hidden Constant

To calculate the hidden constant c, we first isolate s from the left-hand side of (5). Therefore, we use the natural logarithm to decouple s from the exponent position:

$$s \geq \frac{\ln(\frac{1}{3})}{\ln(1 - \frac{\epsilon \cdot d}{2})}. \tag{6}$$

In addition, we isolate c from the right-hand side of (2):

$$c = \frac{s}{\frac{1}{\epsilon \cdot d}}. \tag{7}$$

To find a suitable integer for c, we use the following example, in which one use case is presented with a higher value for ϵ, and another one with ϵ being a very small number. From Lemma 3.2 we know that $\epsilon \cdot d < 1$, since otherwise the input graph can be directly considered ϵ-close to being connected.

Example 1. *Let $\epsilon = 0.5$ and $d = 2$, then by inserting these values into (6) we obtain*

$$s \gtrsim 1.585,$$

Using the calculated value of s in the right-hand side of (7) we get

$$c \gtrsim 1.585.$$

This means that we should set $c \geq 2$.

Now, let $\epsilon = 0.000001$ and keep $d = 2$. By inserting these values into (6) we obtain

$$s \geq 1098612.$$

Using the calculated value of s in the right-hand side of (7) we get

$$c \gtrsim 2.1972.$$

This means that we should set $c \geq 3$.

The two use cases from Example 1 show that the lower bound for c does not vary strongly with decreasing ϵ or d. Thus, for a success probability of at least $2/3$ in identifying a vertex within a small connected component, we should set $c \geq 3$. This holds for a considerable amount of cases.

The analysis of runtime and query complexity of Algorithm 4.1 is provided in Subsection 5.1 after the following presentation of the improved connectivity tester.

4.3 Improved Tester for Connectivity

As suggested in Subsection 3.2, in some cases we can further improve the efficiency of Algorithm 4.1 by dividing the small connected components into intervals of specific component sizes. Therefore, we use the statements of Corollary 3.1.1 and Theorem 3.3 to come up with the following algorithm, which is likewise based on Bhattacharyya & Yoshida [1].

Unlike the first algorithm, Algorithm 4.2 consists of two loops to iterate over the range of intervals of small connected component sizes. Therefore, the first of the two loops iterates over integer values of $[\ell]$, where iterator i ranges from 1 to ℓ, inclusive. As i increases, the number of iterations of the nested second loop decreases, while the vertex threshold at which the BFS algorithm terminates increases. Note that similar to the Theta term in Algorithm 4.1, there is a hidden constant within the Theta term of Algorithm 4.2 in line 3, which is likewise suppressed due to asymptotic simplification. The subsequent logic of vertex selection, BFS conduction, and component size checking from line 4 to line 7 remains the same as for Algorithm 4.1, except for the partially modified termination condition of the BFS in line 5 with respect to the vertex threshold.

Algorithm 4.2 Improved Connectivity Tester

Input: Graph G

Output: Boolean

1: $\ell \coloneqq \lceil log(\frac{2}{\epsilon \cdot d} + 1) \rceil$
2: **for** $i = 1$ to ℓ **do**
3: **for** $t_i \coloneqq \Theta\Big(\frac{\ell}{2^i \cdot \epsilon \cdot d}\Big)$ **do**
4: Select a vertex $v \in V$ uniformly at random.
5: Conduct a BFS from v on, stop when reaching 2^i vertices or no new vertices can be reached.
6: **if** connected component found (of size $< n$) **then**
7: **return** false.
8: **end if**
9: **end for**
10: **end for**
11: **return** true.

4.3.1 Proof of Correctness

To prove the correctness of Algorithm 4.2 in terms of being a one-sided error tester for connectivity, we come up with the following lemma.

Lemma 4.2. *Algorithm 4.2 accepts any connected input graph G and rejects with probability at least $2/3$ if G is ϵ-far from being connected. Thus, Algorithm 4.2 is a one-sided error tester for connectivity.*

Proof. Algorithm 4.2 always returns true if an input graph G is connected, since no small connected components can be found within a connected graph. Thus, it accepts any connected input graph.

Conversely, if G is ϵ-far from being connected, the input graph must be rejected with probability at least $2/3$. As in the proof of Lemma 4.1, this probability also depends on the hidden constant c, which is suppressed in the Theta term of t_i due to asymptotic simplification. Therefore, in the same manner, we refer to the total number of iterations for the second loop as

$$s_i \coloneqq \frac{\ell}{2^i \cdot \epsilon \cdot d} \cdot c, \tag{8}$$

so that c is not suppressed as in the right-hand side of t_i. The following derivation shows the realization of the probability of success and how it is related to the number of iterations s_i per connected component size interval.

We know that the lower bound in terms of contained vertices in each of the connected component intervals of $[\ell]$ is 2^{i-1}. Therefore, we derive the probability of sampling a vertex v contained in a connected component belonging to C_i, which is at least

$$\frac{2^{i-1} \cdot |C_i|}{n}.$$

Substituting $|C_i|$ with the right-hand side of the inequality relation from Theorem 3.3 gives

$$\frac{2^{i-1} \cdot \frac{\epsilon \cdot d \cdot n}{2 \cdot \ell}}{n} = 2^{i-1} \cdot \frac{\epsilon \cdot d}{2 \cdot \ell} = \frac{2^{i-2} \cdot \epsilon \cdot d}{\ell}. \tag{9}$$

To derive the probability of sampling a vertex v that is contained in any set of connected components C_i, we include the total number of iterations per set, s_i, into the inequality. Therefore, we first include the hidden constant c into (9):

$$\frac{2^{i-2} \cdot \epsilon \cdot d}{\ell} \geq \frac{2^{i+1} \cdot \epsilon \cdot d}{\ell \cdot c}, \text{ where } c \geq 2^3, \tag{10}$$

so that a part of the expression can easily be substituted to s_i. In doing so, we arrive at the aforementioned probability, which is at least

$$\frac{2^{i+1} \cdot \epsilon \cdot d}{\ell \cdot c} \geq \frac{2}{s_i}, \text{ where } c \geq 2^3.$$

Note that we choose $c \geq 8$ to be able to substitute the expression with s_i. Then, we derive the probability of not sampling a vertex contained in C_i, which is less than

$$1 - \frac{2}{s_i}. \tag{11}$$

Furthermore, we use (11) to obtain the probability of not sampling a vertex that is contained in any set of connected components C_i, that is less than

$$\left(1 - \frac{2}{s_i}\right)^{s_i}. \tag{12}$$

Then, we form the complementary probability of (12), which is the probability of sampling a vertex that is contained in any set of connected components C_i, which is at least

$$1 - \left(1 - \frac{2}{s_i}\right)^{s_i} > \frac{2}{3},$$

whereby $c \geq 8$. Hence, we can state that Algorithm 4.2 is a one-sided error tester for connectivity.

QED

4.3.2 Calculating the Hidden Constant

To determine the hidden constant c for the improved tester, we analyze the sampling process on each interval $i \in [\ell]$. As stated in the proof of Lemma 4.2, the probability of selecting a vertex belonging to a component from C_i in a single trial is at least

$$p_i = \frac{2^{i-2} \cdot \epsilon \cdot d}{\ell}.$$

If s_i independent trials are performed for interval i, the probability that none of them hits such a vertex is bounded by

$$(1 - p_i)^{s_i}.$$

To guarantee detection within interval i with probability at least $2/3$, we require

$$(1 - p_i)^{s_i} \leq \tfrac{1}{3}.$$

Taking natural logarithms and solving for s_i yields

$$s_i \;\geq\; \frac{\ln(1/3)}{\ln(1 - p_i)}. \tag{13}$$

Next, we relate s_i to the hidden constant c. From the explicit expression for t_i in Algorithm 4.2, and the definition of the number of trials per interval (see (8)), we obtain

$$s_i \;=\; \frac{\ell}{2^i \cdot \epsilon \cdot d} \;\cdot c,$$

which gives the following lower bound on c:

$$c \;\geq\; s_i \cdot \frac{2^i \cdot \epsilon \cdot d}{\ell}. \tag{14}$$

Example 2. *We demonstrate how to calculate the hidden constant c for the improved connectivity tester using two sets of parameters: one with a large value of ϵ and one with a small value of ϵ.*

- ***Case 1:*** $i \in \{1, 2\}$, $\ell = 2$, $\epsilon = 0.5$, $d = 2$.

$$p_1 = \frac{2^{1-2} \cdot 0.5 \cdot 2}{2} = 0.25,$$

$$s_1 = \frac{\ln(1/3)}{\ln(1 - 0.25)} = \frac{1.098612289}{\ln 0.75} \approx 3.822,$$

$$c_1 = s_1 \cdot \frac{2^1 \cdot 0.5 \cdot 2}{2} = 3.822 \cdot 1 \approx 3.822.$$

$$p_2 = \frac{2^{2-2} \cdot 0.5 \cdot 2}{2} = 0.5,$$

$$s_2 = \frac{\ln(1/3)}{\ln(1-0.5)} = \frac{1.098612289}{\ln 0.5} \approx 1.585,$$

$$c_2 = s_2 \cdot \frac{2^2 \cdot 0.5 \cdot 2}{2} = 1.585 \cdot 2 \approx 3.17.$$

Taking the maximum over c_i gives

$$c = \max(c_1, c_2) \approx 3.822 \implies c \geq 4.$$

- ***Case 2:*** $i \in \{1, 2, \ldots, 20\}$, $\ell = 20$, $\epsilon = 10^{-6}$, $d = 2$.

$$p_i = \frac{2^{i-2} \cdot 10^{-6} \cdot 2}{20} = 2^{i-2} \cdot 10^{-7}.$$

Illustrative values:

$$p_1 = 2^{-1} \cdot 10^{-7} = 5 \cdot 10^{-8}, \qquad p_{20} = 2^{18} \cdot 10^{-7} \approx 0.0262144.$$

For small p_i, we can approximate $\ln(1 - p_i) \approx -p_i$, giving

$$s_i \approx \frac{\ln(1/3)}{p_i} \approx \frac{1.098612289}{p_i}, \quad c_i \approx s_i \cdot \frac{2^i \epsilon d}{\ell}.$$

With this approximation, the c_i values are nearly constant across i, e.g.,

$$c_1 \approx 4.394, \quad c_{20} \approx 4.394.$$

Taking the maximum over all $i \in [20]$:

$$c = \max_i c_i \approx 4.394 \implies c \geq 5.$$

These two cases from Example 2 illustrate that the required constant c depends on the parameters ϵ, d, i, and ℓ. For moderate ϵ, c varies noticeably with i and p_i, whereas for very small ϵ, the c_i values are nearly uniform across all intervals. In all cases, we take $c = \max_i c_i$ as the final hidden constant, and in practice, one can select a conservative uniform value of c based on the smallest expected ϵ and the range of intervals to ensure sufficient samples in each interval.

The analysis of the runtime and query complexity of Algorithm 4.2 is presented in Subsection 5.2, while Subsection 5.3 provides a comparison of those metrics between the two algorithms and shows the efficiency improvement of Algorithm 4.2 in detail.

5 Analysis and Comparison of the Algorithms

5.1 Analysis of Algorithm 4.1

5.1.1 Query Complexity

In the case of property testing on graphs with query access, the query complexity is a key factor that makes related algorithms, e.g. those with the same time complexity, comparable. We derive the query complexity by considering the worst-case scenario of possible queries that we have to perform on an input graph G [5, 14].

Here, queries are performed when conducting a BFS, as in line 3 of Algorithm 4.1. We stop the BFS when the threshold of vertices for small connected components is reached, which in the worst case requires d queries for each vertex. This leads to a query complexity of

$$\mathcal{O}\left(\frac{2}{\epsilon \cdot d} \cdot d\right) = \mathcal{O}\left(\frac{2}{\epsilon}\right) \tag{15}$$

per iteration. In the worst case, G is a connected graph, so we have

$$\mathcal{O}\left(\frac{1}{\epsilon \cdot d} \cdot c\right) = \mathcal{O}\left(\frac{1}{\epsilon \cdot d}\right) \tag{16}$$

iterations in total. We obtain the total query complexity of Algorithm 4.1 by multiplying (15) and (16):

$$\mathcal{O}\left(\frac{2}{\epsilon}\right) \cdot \mathcal{O}\left(\frac{1}{\epsilon \cdot d}\right) = \mathcal{O}\left(\frac{1}{\epsilon^2 \cdot d}\right). \tag{17}$$

5.1.2 Runtime Analysis

When analyzing Algorithm 4.1, there are two aspects that can influence the runtime. Similar to the query complexity, these are the threshold value used for the BFS termination and the total number of iterations, denoted by t. The main difference compared to the basic approach presented in Subsection 2.3 is that the total number of vertices, n, is neither included in the termination condition of the BFS nor in the iteration count, at least not as part of their calculation. Since the other two parameters, ϵ and d, are explicitly defined constants, the runtime of Algorithm 4.1 remains constant. Having established these theoretical efficiency gains, we now turn to empirical validation to assess how these asymptotic improvements translate to practical performance.

5.2 Analysis of Algorithm 4.2

5.2.1 Query Complexity

For Algorithm 4.2, we determine the query complexity in a similar way as we did for Algorithm 4.1 in Subsection 5.1. The differences here involve the dynamic termination condition of the BFS, which depends on the iterator $i \in [\ell]$ of the outer loop, and the fact that we have two loops to consider instead of just one. In the worst case, given a connected graph G where all vertices have d neighbors, we end up with a query complexity of

$$\mathcal{O}\left(\sum_{i=1}^{\ell} \frac{\ell}{2^i \cdot \epsilon \cdot d} \cdot 2^i \cdot d\right) = \mathcal{O}\left(\sum_{i=1}^{\ell} \frac{\ell}{\epsilon}\right). \tag{18}$$

Since the control variable of the sigma term in the right expression of (18) is no longer tied to any term, we can further reduce (18) to

$$\mathcal{O}\left(\ell \cdot \frac{\ell}{\epsilon}\right) = \mathcal{O}\left(\frac{\ell^2}{\epsilon}\right).$$

As a last step, we can substitute ℓ and obtain the total query complexity of Algorithm 4.2:

$$\mathcal{O}\left(\frac{\log^2(\frac{1}{\epsilon \cdot d})}{\epsilon}\right). \tag{19}$$

Note that the ceiling function and the numerator of value two in ℓ are suppressed within the Omega expression due to asymptotic simplification.

5.2.2 Runtime Analysis

As for the runtime analysis of Algorithm 4.1, the threshold value used for the BFS termination and the total number of iterations of both contained loops could potentially have the most impact on the runtime. But since the total number of vertices n is neither included in the iteration counts of the loops nor in the BFS termination condition, and since ϵ and d are likewise fixed constants, the runtime of Algorithm 4.2 remains constant as well.

5.3 Metric-Based Comparison of the Algorithms

The runtimes of both presented algorithms are already constant, so our focus is mainly on comparing the query complexities. By examining the query complexity of Algorithm 4.1, we notice that the denominator contains ϵ squared, while this is not the case for the query complexity of Algorithm 4.2. Consequently, as ϵ decreases, the resulting query complexity of Algorithm 4.2 increases at a much slower rate compared to the query complexity of Algorithm 4.1. Note that we can reduce the query complexity of Algorithm 4.2 even further, since the numerator inside the expression has only logarithmic growth, which has a negligible impact on the overall efficiency [5]. Therefore, we use the soft-Omega notation, which reduces the query complexity of Algorithm 4.2

to

$$\widetilde{\mathcal{O}}\Big(\frac{1}{\epsilon}\Big) = \widetilde{\mathcal{O}}\Big(\epsilon^{-1}\Big). \tag{20}$$

This reduction also helps to emphasize the improvement over the query complexity of Algorithm 4.1. To illustrate this, we use the following example.

Example 3. *Let $\epsilon = 0.01$ and $d = 10$, where $n > d$. First, we calculate the query complexity (QC) of Algorithm 4.1:*

$$QC_{\textit{Algorithm 4.1}} = \frac{1}{0.01^2 \cdot 10} = 1000.$$

Using (20) we calculate the query complexity of Algorithm 4.2:

$$QC_{\textit{Algorithm 4.2}} = 0.01^{-1} = 100.$$

Now, let $\epsilon = 0.001$, while keeping the other conditions the same. We recompute the query complexities for both algorithms:

$$QC_{\textit{Algorithm 4.1}} = \frac{1}{0.001^2 \cdot 10} = 100\,000,$$

$$QC_{\textit{Algorithm 4.2}} = 0.001^{-1} = 1000.$$

From Example 3, we observe that Algorithm 4.2 has a query complexity that improves over that of Algorithm 4.1 by a factor of $\frac{1}{10 \cdot \epsilon}$ for decreasing values of ϵ, when neglecting logarithmic factors. The smaller the value of ϵ, the greater the efficiency advantage of Algorithm 4.2 over Algorithm 4.1 in terms of query complexity. From Lemma 3.2 we know that $\epsilon \cdot d < 1$ for any input graph G, otherwise G can directly be considered as ϵ-close to being connected. Taking this into account, for small values of ϵ, Algorithm 4.2 offers a significant advantage in terms of query complexity compared to Algorithm 4.1.

6 Implementation and Empirical Evaluation

This section presents an implementation-based study of the connectivity testers from Section 4, together with a baseline plain BFS connectivity check, with all code available in our Git repository [1]. The goal is to evaluate how the theoretical guarantees manifest in practice, to understand how parameter choices influence efficiency, and to identify scenarios in which the sublinear testers

[1] https://github.com/penthooose/connectivity_testing_bounded_degree

are preferable to a full traversal of the graph.

All experiments were conducted on synthetically generated bounded-degree graphs of size up to ten million vertices. The graphs were constructed to include heterogeneous component sizes. While many real-world networks exhibit heavy-tailed characteristics (e.g., in their degree distributions), component-size distributions can also show substantial variation [10]. Our synthetic graphs reflect this heterogeneity by including many small components and a few large ones. For each instance we recorded component sizes, execution times, iteration counts, and the number of visited vertices.

6.1 Experimental Setup

We implemented:

- a plain BFS connectivity tester (full traversal from a randomly chosen start vertex),
- the simple tester of Algorithm 4.1, and
- the improved interval-based tester of Algorithm 4.2.

All implementations strictly follow the bounded-degree model: adjacency queries are constant-time, and uniform random vertex sampling is available. For the sublinear testers, the parameters $1/(\epsilon d)$ and the hidden constants appearing in the theoretical bounds were exposed explicitly. This allowed us to investigate how parameter selection affects actual running time and query behavior.

6.2 Baseline: Plain BFS Behaviour

Although BFS is linear in the size of the explored component, its running time in practice varies substantially depending on the start vertex. For graphs with large components, BFS may traverse a significant fraction of all n vertices. In our experiments for graphs around two million vertices, we observed both relatively fast runs (around one second) and much slower runs (around six seconds), depending on which component the starting vertex lies in.

Representative examples illustrate this variance:

- On a graph with 20 components, including several small ones, BFS terminated after visiting roughly 1.7×10^5 vertices.
- On a graph with only 5 components, the BFS from a typical starting vertex often covered more than one million vertices.

Because BFS explores entire components, its runtime is heavily correlated with the tail of the component size distribution. This motivates the use of property testers when one expects a graph to have many small components.

6.3 Behavior of the Simple Tester (Algorithm 4.1)

The simple tester performs a bounded BFS to depth $2/(\epsilon d)$ and repeats this process $c/(\epsilon d)$ times. In all experiments, this resulted in strictly sublinear exploration: at most $15{,}000$ visited vertices for $\epsilon = 10^{-3}$ and $d = 20$.

However, the empirical results highlight an important issue: many tested graphs are not ϵ-far from connected for typical ϵ values. For example, in a graph of $n = 2 \times 10^6$ and $d = 20$, one requires

$$\epsilon < \frac{c}{dn} \approx 5 \times 10^{-7}$$

for a 20-component instance to be classified as ϵ-far. If one instead uses $\epsilon = 10^{-3}$, then theoretically the graph is considered ϵ-close, and the tester has no reason to find a small component. Indeed, the tester accepts quickly, visiting only a small number of vertices.

This demonstrates that choosing ϵ too large makes the tester insensitive to the component structure, while choosing it too small increases the number of iterations and BFS depth to the point where the procedure may be more expensive than a plain BFS.

6.4 Behavior of the Improved Tester (Algorithm 4.2)

The improved tester partitions possible component sizes into intervals

$$[1], [2, 3], [4, 7], [8, 15], \ldots,$$

and chooses iteration counts adapted to the likelihood of encountering a component in each range.

Empirically, this reduces unnecessary work in the small-size areas: the tester focuses many samples on tiny components, which are the ones that matter for detecting that a graph is ϵ-far from connected. For graphs with many small components, this behavior provides a pronounced advantage over both plain BFS and Algorithm 4.1.

6.5 Comparison with Experiment Logs

Across all tested graphs and algorithms, the main observations were:

1. **Plain BFS** is fast when started in a small component, but can be slow when the start vertex lies in a large component.
2. **Algorithm 4.1** is consistently fast, but only useful when the graph is actually ϵ-far under the chosen parameters.
3. **Algorithm 4.2** outperforms both BFS and Algorithm 4.1 when the graph contains many very small components and the goal is to detect one of them via uniform random sampling.

These results align with the intuition that the advantage of sublinear testers comes from hitting a small component quickly. When most components are large, or when ϵ is too large for the instance

to qualify as ϵ-far, the testers cannot significantly improve on plain BFS.

6.6 Practical Guidelines and Implications

From the experiments, several practical lessons emerge:

- **Understanding the application domain is essential.** Different domains (e.g. social networks, molecular structures, genomic interaction graphs) produce very different component size distributions [2, 10]. Knowing whether many small components are expected determines whether a sublinear tester is effective.
- **The choice of ϵ is crucial.** A too-small ϵ increases the iteration count and BFS budget, making the tester potentially more expensive than BFS. A too-large ϵ makes the tester incapable of detecting disconnectedness. Thus, ϵ must match the expected component structure.
- **Power-law or heavy-tailed component-size distributions favor sublinear testers.** When the graph contains a large number of very small components and only a few large ones, random sampling finds a small component quickly, making Algorithm 4.2 especially efficient.
- **If the graph lacks many small components, plain BFS may be preferable.** In such cases the theoretical ϵ-far criterion essentially forces ϵ to be so small that the tester explores a large fraction of the graph.

7 Conclusion

Overall, this work presented two algorithms that achieve constant runtime. Both algorithms were proven to be one-sided connectivity testers, even though both of them involve the incorporation of a hidden constant in order to fulfill the requirement of being a tester. Moreover, the algorithms are designed for the general domain of bounded-degree graphs, meaning that all possible graphs within this class are taken into account. However, in real-world applications with specific graph properties, minor adjustments to the algorithms, such as advanced termination conditions, can help to achieve optimal accuracy.

Furthermore, practical implementations of these algorithms may face additional implementation-related issues that can significantly impact time, space, or query complexities. For instance, there may be differences in data structures or functions, and more substantially, limitations on resources like computing power or memory. Therefore, analyzing and optimizing the individual implementation remains essential, as shown by Weise et al. [13].

The empirical evaluation carried out in this work highlights how important it is to understand the graph structures that arise from specific applications. Different domains can lead to very different component-size distributions, which in turn influence how well the testers perform. The experiments showed that the benefits of sublinear connectivity testing become most apparent when a graph contains many small components, since in such cases a random starting point has a reason-

able chance of hitting one of them early. When this structural property is missing, or when ϵ is not chosen carefully, the iteration schedule can dominate the workload, and a plain BFS may end up being the simpler and faster option in practice.

While this paper reviews efficient connectivity testing techniques for the bounded-degree model, there are conceptual successors that build on this groundwork. For example, there are efficient algorithms for testing k-edge and k-vertex connectivity, described in the works of Bhattacharyya & Yoshida [1] and Goldreich & Ron [4].

Ultimately, the field of property testing has immense potential for the future. With the growing importance of graphs in modern applications, continued research is essential. This will not only lead to more efficient algorithms, but also ensure that these techniques can handle the ever-increasing size and complexity of graphs arising from real-world use cases.

References

[1] Arnab Bhattacharyya and Yuichi Yoshida. *Graphs in the Bounded-Degree Model*, pages 103 – 144. Springer Singapore, Singapore, 2022.

[2] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[3] W.G. Cochran. *Sampling Techniques*, pages 1 – 44. Wiley Series in Probability and Statistics. Wiley, 1977.

[4] Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 406 – 415, New York, NY, USA, 1997. Association for Computing Machinery.

[5] Oded Goldreich. *A Brief Introduction to Property Testing*, pages 1–5. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[6] Oded Goldreich. *Introduction to Testing Graph Properties*, pages 105 – 141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[7] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78 – 94, 7 2018.

[8] Jason Holdsworth. The nature of breadth-first search. *School of Computer Science, Mathematics, and Physics, James Cook University, Australia, Tech. Rep*, pages 99 – 112, 1999.

[9] Basudeb Mondal and Kajal De. An overview applications of graph theory in real field. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 2(5):751 – 759, 2017.

[10] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[11] Karthik Raman. Construction and analysis of protein–protein interaction networks. *Automated experimentation*, 2:2, 02 2010.

[12] Ronitt Rubinfeld and Asaf Shapira. Sublinear time algorithms. *SIAM Journal on Discrete Mathematics*, 25(4):1562 – 1588, 2011.

[13] Thomas Weise, Yuezhong Wu, Weichen Liu, and Raymond Chiong. Implementation issues in optimization algorithms: do they matter? *Journal of Experimental & Theoretical Artificial Intelligence*, 31(4):533 – 554, 2019.

[14] Herbert S. Wilf. *Mathematical Preliminaries*, pages 9 – 39. A K Peters/CRC Press, New York, 2nd edition, 2002. eBook ISBN: 9780429294921.

