TEL AVIV UNIVERSITY
The Iby and Aladar Fleischman Faculty of Engineering
The Zandman-Slaner School of Graduate Studies

ON FINDING SMALL SUBGRAPHS IN BOUNDED-DEGREE GRAPHS

A thesis submitted toward the degree of
Master of Science in Engineering

by

Yaniv Sabo

November 2016

TEL AVIV UNIVERSITY
The Iby and Aladar Fleischman Faculty of Engineering
The Zandman-Slaner School of Graduate Studies

ON FINDING SMALL SUBGRAPHS IN BOUNDED-DEGREE GRAPHS

A thesis submitted toward the degree of
Master of Science in Engineering

by

Yaniv Sabo

This research was carried out at Tel Aviv University
in the Department of Electrical Engineering - Systems
Faculty of Engineering
under the supervision of Prof. Dana Ron

November 2016

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Dana Ron. Her patience, immense knowledge and guidance were beyond compare and I am truly grateful.

I would also like to thank my family and friends, for supporting me spiritually throughout writing this thesis and my life in general.

Abstract

We study the problem of finding a copy of a subgraph H in a graph G that is far from being free of having copies of H. We consider this problem in the bounded-degree graphs model. In this model, each of the n vertices has at most d neighbors, and an algorithm is allowed to make queries regarding the neighbors of vertices in the graph. The graph is said to be ϵ-*far* from being H-free if more than $\epsilon \cdot dn$ of its edges must be deleted to make the graph free from having copies of H.

We present an algorithm for finding a copy of H in graphs that are ϵ-far from being H-free and have bounded (constant) treewidth. This algorithm makes a number of queries that is polynomial in $1/\epsilon$, the size of H and the degree bound d. The complexity of the algorithm is independent of the number of vertices, n.

We also present an algorithm for the special case in which H is a path of length k. Our algorithm uses specific properties of graphs that are far from having k-paths. Finding k-paths was previously studied by Reznik (*Master's thesis, Weizmann Institute of Science, 2011*). Reznik gave an algorithm for the case in which G is cycle-free, where the query complexity of the algorithm is polynomial in k, d and $1/\epsilon$. We propose a conjecture that, if proven to be true, implies that our algorithm works for any graph that is ϵ-far from being k-path free with query complexity polynomial in k, d, and $1/\epsilon$. As a sanity check, we establish the conjecture for cycle-free graphs.

Table of Contents

List of Figures

1 Introduction

1.1 Property Testing

Property testing is the study of highly efficient algorithms that decide whether a given object (e.g., a graph) has a certain predetermined property (e.g., bipartiteness), or is significantly different from any object that has the property. The algorithm is given access to an oracle for performing (local) queries to the input and a small probability of failure is allowed. Property testing was first explicitly defined in the work of Rubinfeld and Sudan [RS96], who considered testing whether a function is a low-degree polynomial. A systematic study of property testing with special focus on graph properties was undertaken in [GGR98].

In order to define a property testing problem, we need to describe the type of queries that the algorithm is allowed to perform, provide a distance measure between objects and define the class of objects for which the property holds. The algorithm is given a *distance* parameter, denoted ϵ, and should *accept* inputs (with probability at least 2/3) for which the property holds and *reject* inputs (with probability at least 2/3) which are ϵ-far from any other object for which the property holds. If the input neither has the property nor is far from having it, then the algorithm can either accept or reject. In the following sections, when we use the term *high probability*, we mean high constant probability, i.e., probability at least 2/3. In some cases testing algorithms have a non-zero error probability only on inputs that are far from having the property. In such a case, when they reject an input, they always provide evidence that the input does not have the property.

The main complexity measure for property testing algorithms is their *query complexity.* The aim is to design testing algorithms whose query complexity is sub-linear in the size of the tested object, and possibly even independent of this size.

The study of *testing graph properties* was initiated by Goldreich, Goldwasser and Ron [GGR98]. A *graph property* is a set of graphs closed under graph isomorphism (renaming of vertices). When studying graph properties, the form of queries, and in some cases the distance measure, depend on the graph representation. In this work we are interested in bounded degree graphs, represented by adjacency lists. We describe the precise model shortly.

1.2 Testing for Subgraph Freeness

In this thesis, we study the problem of testing H-freeness for a given small subgraph H. A graph G has the property of H-freeness if there is no subgraph of G that is isomorphic to H. We study the problem in the *bounded-degree incidence-lists model* presented in [GR97]. In this model, a tester for H-freeness is a randomized algorithm, which gets as input a degree bound, d, a distance parameter, ϵ, and the number of vertices in the graph, n. The tester is given access to an oracle, allowing it to make queries regarding the graph. These queries are of the form (v, i) where v is a vertex in the graph and i is a number between 1 and d, with the answer being the i-th neighbor of v (if v has at least i neighbors, otherwise a special symbol is returned). The algorithm should distinguish with probability at least 2/3 between the case in which G has no copy of H and the case in which it is necessary to remove at least ϵdn edges from the graph for there to be no remaining copies of H in G. In the first case the tester should *accept* G, and in the second case it should *reject* G. We are interested in testers that perform a sub-linear number of queries with respect to the size of the graph. We shall actually design algorithms for which the number of queries is independent of the number of vertices in the graph, and depends only on the parameters ϵ, d, the size of H, and possibly on properties of H. As we show in Subsection 3.1, we can assume without loss of generality that H is connected.

Specifically, we will be interested in algorithms that have *one-sided* error. Namely, the tester must accept the graph with probability 1 if it is H-free. This means that in order to reject a graph the tester must find evidence of the subgraph H in G, since otherwise it might reject graphs that are H-free. Therefore, another way of stating the problem is of finding a copy of H in G (with high probability), when G is ϵ-far from being H-free. We will give special consideration to the case in which H is a k-path, i.e., k vertices $v_1, v_2, ..., v_k$ such that there is an edge between v_i and v_{i+1} for $1 \leq i \leq k-1$.

1.3 A Simple Algorithm for Finding Subgraphs

As observed in [GR97], if G is ϵ-far from being H-free, then it is trivial to find, with high success probability, a subgraph H in G using $O(d^{\tau+1}/\epsilon)$ queries, where τ is the diameter of H. Notice that there should exist at least ϵn vertices that participate in some copy of H. Otherwise, we can just remove all edges incident to such vertices, so that we are left with no copy of H in the graph, while removing less than ϵdn edges. Hence, if we choose a vertex in the graph with uniform probability and perform a *BFS* up to depth τ, we will find a copy of H with probability at least ϵ. By executing this $O(1/\epsilon)$ times, we will find evidence of H with high probability, performing a total of $O(d^{\tau+1}/\epsilon)$ queries. Notice that using this method, we get query complexity that is independent of the size of the input graph, but exponential in τ. We are interested in the question of whether it is possible to find a subgraph H with query complexity polynomial in d, $1/\epsilon$ and the size of H when G is ϵ-far from being H-free.

1.4 Previous Work

The specific problem of finding a k-path in graphs that are far from k-path free, was first addressed by Czumaj et al. [CGR$^+$12], as a simple special case of finding an M-minor (in graphs that are far from M-minor free) for M that is a tree. A graph G is *M-minor free* for a given graph M, if M cannot be obtained from G by a sequence of vertex removals, edge removals and edge contractions.[1] (We give more details on previous work related to testing minor freeness in Subsection 1.6.) Observe that while finding subgraphs and finding minors are very different problems in general, in the case of a k-path, the problem of finding a k-path minor is the same as finding a k-path. Czumaj et al. [CGR$^+$12] raised the question whether there is an algorithm for finding a k-path in bounded-degree graphs that are far from k-path free whose query complexity is poly$(d, k, 1/\epsilon)$. Note that the desired bound is on the query complexity and not the running time, since by setting $d = n$, $k = n$ and $\epsilon = 1/n^2$, we would get an algorithm for finding a Hamiltonian path (if one exists).

Reznik studied this question in [Rez11]. He shows that it is possible to find (with high probability) a k-path in a *cycle-free graph* that is ϵ-far from being k-path free,

[1] An edge contraction is an operation that removes an edge from the graph and merges the two vertices that it previously connected. Namely, an edge (u, v) is replaced by a single vertex u', where every vertex w that was a neighbor of either u or v, is now a neighbor of u'.

with query complexity polynomial in k, d and $1/\epsilon$. He gives two algorithms (random DFS and random walks) for finding k-paths. The random DFS algorithm has query complexity $O\left(\frac{dk^2}{\epsilon^4}\right)$ and the random walk algorithm has query complexity $O\left(\frac{d^{15}k^7}{\epsilon^4}\right)$. The analysis of these algorithms (on cycle-free graphs) is based on first establishing the existence of a special subgraph (in which it is relatively simple to find a k-path) in cycle-free graphs that are ϵ-far from being k-path free. Reznik then shows that both searches, given they had started from a vertex in this subgraph, generally stay in it, while occasionally straying from the subgraph.

1.4.1 Partition Oracles

There has been some previous work on designing *partition oracles* for various families of graphs. These procedures are also given query access to the incidence lists representation of a bounded-degree graph and a distance parameter ϵ'. A partition oracle is defined with respect to a class of graphs, $\mathcal{C}$. When the procedure is queried on a vertex in the graph, it returns the part (subset of vertices) to which the vertex belongs in a partition of the graph vertices. The partition should be such that the parts are small (relative to $1/\epsilon'$) and if the graph belongs to $\mathcal{C}$, with high probability, the total number of edges between parts is at most $\epsilon'|V|$. In particular, there has been work done on designing partition oracles for graphs that are free from some constant size minor, M.

One can make use of a partition oracle to find a subgraph H in a graph G, under the premise that G is free from some minor M (of constant size) and given that G is ϵ-far from being H-free. This can be accomplished by uniformly selecting a set of starting vertices and performing a local search for H from each of them, while limiting the number of steps. During the walk, the partition oracle procedure is called in order to make sure that the walk does not leave the part it started at. Intuitively, because there aren't many edges between parts, then there should be many vertices that belong to a part with copies of H contained entirely in it. If we start the search at one of these parts and use the oracle to make sure we do not leave it, we will be able to find a copy of H while visiting a small number of vertices, since the parts are relatively small.

In [LR13] a quasi-polynomial partition oracle was given for graphs with an excluded minor M. We can use this to construct a quasi-polynomial (in $1/\epsilon$, d and the size of H, assuming constant size M) algorithm for finding H in graphs with an excluded minor M which are ϵ-far from being H-free.

In [EHNO11] a partition oracle was designed for the class of graphs with bounded-treewidth (which will be defined in Section 2). Assuming that the bound on the treewidth is constant, the oracle given in [EHNO11] has complexity poly(d/ϵ). There are known families of graphs with bounded treewidth, such as cactus graphs, outerplanar graphs and pseudoforests. On the other hand, there are families of graphs with an excluded minor that do not have bounded treewidth, such as planar graphs.

1.5 Our Main Results

Our first result is an algorithm that, given query access to a cycle-free graph G with maximum degree d, tests for H-freeness (with one-sided error). The query complexity of the algorithm is polynomial in d, $1/\epsilon$ and the size of H. We take a different approach from the one appearing in [Rez11], allowing us to find general subgraphs (instead of just k-paths) and give a relatively simple proof that the algorithm works for cycle-free graphs. The query complexity of our algorithm for k-paths in cycle-free graphs is similar to what is shown in [Rez11]. We then extend this result to bounded-treewidth graphs (which will be defined in Section 2). We do so by making use of results appearing in [EHNO11], in which an efficient partition oracle is given for bounded-treewidth graphs. As previously explained, it is possible to make a more direct, black-box use of the partition oracle to give an algorithm for finding H in bounded-treewidth graphs which are ϵ-far from being H-free. However, we have chosen to build on a lemma from [EHNO11] to show that our simple and more efficient algorithm works for bounded-treewidth graphs as well. We also use a procedure (Procedure 1) that is similar to to a procedure that appears in [EHNO11] (Algorithm 2).

The second part of this thesis focuses on k-paths. Our original goal was to obtain a one-sided error algorithm for testing k-path freeness in general graphs, whose query complexity is polynomial in d, k, and $1/\epsilon$. While this goal was not achieved, we made a step that we hope will lead to achieving this goal in the future. We start by presenting a basic property of DFS walks for which the set of vertices visited during the DFS induces a subgraph that does not contain any k-path. We then state a conjecture regarding a certain property of random DFS walks. This conjecture allows us to design a framework for a proof which, given that this conjecture is proven true, shows that the random DFS algorithm indeed finds a k-path with high probability in general bounded-degree graphs that are ϵ-far from being k-path free.

As a sanity check for the viability of this conjecture we prove its correctness for the case of random DFS walks in cycle-free graphs. The question of the correctness of this conjecture in general bounded-degree graphs is left open.

Note that we are only referring to the query complexity of the algorithms we present in this thesis, and not the running time. As observed in Subsection 1.4, when $d = n$, $k = n$ and $\epsilon = \frac{1}{n^2}$ finding a k-path is equivalent to finding a Hamiltonian path. The general approach we took involves traversing a subset of vertices S, querying the edges incident to each of them, and then finding a k-path in the subgraph induced by S. The task of finding a k-path in S might require exponential running time (in terms of k, d and $1/\epsilon$) but this does not affect the query complexity, which only depends on the size of S.

1.6 Other Related Work

1.6.1 Minors

The problem of testing minor-freeness for any given constant size minor M in bounded-degree graphs, was first studied by Benjamini, Schramm, and Shapira [BSS08]. They showed a general result for testing whether a graph holds some monotone [2] hyper-finite [3] graph property. We focus here on a corollary of this result, for minor free graphs. They gave a two-sided error testing algorithm whose query complexity for constant degree graphs and constant size M is $O\left(2^{d^{d^{\mathrm{poly}(d/\epsilon)}}}\right)$, i.e., independent of the size of G. This result was later improved (in terms of the dependance on the distance parameter, ϵ, and the degree bound, d) by Hassidim et al. [HKNO09], who obtained a testing algorithm for M-minor freeness whose complexity is $O\left(d^{\mathrm{poly}(1/\epsilon)}\right)$, and later in [LR13], to $d^{O(\log^2(1/\epsilon))}$.

As mentioned earlier, the problem of one-sided error testing of M-minor freeness (i.e., finding a minor in graphs that are far from M-minor free) was considered by Czumaj et al. [CGR$^+$12]. On the negative side they show that for any fixed M that contains a simple cycle, the query complexity of one-sided error testing of M-minor freeness is $\Omega(\sqrt{n})$. On the positive side they give an algorithm that finds a simple

[2] A graph property $\mathcal{P}$ is monotone if every subgraph of a graph in $\mathcal{P}$ is also in $\mathcal{P}$

[3] A graph $G = (V, E)$ is (δ, k)-hyper-finite if one can remove $\delta|V|$ edges from G and obtain a graph with connected components of size at most k. A collection of graphs is hyper-finite if for every $\delta > 0$ there is some finite $k = k(\delta)$ such that every graph in the collection is (δ, k)-hyper-finite

cycle in a graph that is ϵ-far from being cycle free, with expected running time of $\widetilde{O}\left(\text{poly}(d/\epsilon) \cdot \sqrt{n}\right)$. This positive result, which is equivalent to one-sided error testing of C_3-minor freeness, is extended to C_k-minor freeness. They also consider the case of finding a T-minor, where T is a tree (in graphs that are far from being T-minor free), and give an algorithm whose complexity depends only on d, ϵ, and the size of T.

1.6.2 Random DFS Walks

We were interested in analyzing certain properties of random DFS walks, by which we mean a DFS such that in each step of the search, the next vertex is chosen uniformly at random among the set of unvisited neighbors of the current vertex (or backtracking, as usual, in the case of an empty set). However, we were not able to find previous work discussing properties of random DFS walks. There are some works analyzing *non-backtracking random walks*, which are random walks that do not return to the previous vertex visited, such as [ABLS07]. One can consider these as random walks with a memory bank of one *already-visited-vertex*, as opposed to the random DFS walks which are random walks with a memory bank of $|V|$ *already-visited-vertices.*

2 Preliminaries

2.1 Basic Definitions and Notations

We denote by $G = (V, E)$ a simple undirected graph with n vertices, such that each vertex in V has at most d neighbors. Also, we denote by $N(U)$ the set of vertices in $V \setminus U$ that are neighbors of vertices in U, i.e. $N(U) = \{v \in V \setminus U \mid \exists u \in U, (u, v) \in E\}$. For a subset of vertices $S \subseteq V$, we denote by $G(S)$ the subgraph induced by the vertices in S.

2.2 Testing Graph Properties in the Bounded-Degree Model

Let $\mathcal{P}$ be a *property* of graphs. That is, $\mathcal{P}$ defines a subset of graphs, where the subset is closed under graph isomorphism. We say that a graph G has the property if $G \in \mathcal{P}$. In the bounded degree model, a graph is said to be ϵ-far from having a property $\mathcal{P}$ if more than ϵdn edge modifications should be performed on the graph so that it obtains the property. In this case ϵ measures (up to a factor of 2), the fraction of entries in the incidence lists representation of G that should be modified so that $G \in \mathcal{P}$.

In the bounded-degree model, a testing algorithm is allowed to probe the incidence lists of the vertices in the graph. More precisely, the algorithm is given query access to a function $f_G \colon V \times [d] \to V \bigcup \Gamma$ such that $f(v, i) = u$ if u is the i^{th} neighbor of v (according to some arbitrary but fixed ordering of the neighbors), and $f(v, i) = \Gamma$ if v has less than i neighbors. We now spell the meaning of property testing in this model.

Definition 1. *A tester for a graph property $\mathcal{P}$ is a probabilistic algorithm that, given input paramters n, d, ϵ and query access to f_G, outputs a binary verdict that satisfies the following two conditions.*

1. *If $G \in \mathcal{P}$ then the tester accepts with probability at least* $2/3$.

2. *If G is ϵ-far from having the property $\mathcal{P}$ then the tester rejects with probability at least* $2/3$.

If the tester accepts every graph in $\mathcal{P}$ with probability 1, *then we say that it has* one-sided error.

The *query complexity* of a tester is the number of queries it makes, as a function of the parameters n, d, ϵ and other parameters that relate to the tested property.

2.3 Random DFS scans

We use DFS scans to traverse the graph, with two small variations. First, we limit the number of steps the search is allowed to do. Secondly, we scan the graph in a random order, so whenever we reach a vertex we haven't seen before or backtrack to an already visited vertex, we choose the next neighbor to visit randomly from the set of neighbors which were not already visited. Unless stated otherwise, when we say that a DFS performed s steps, we mean that it visited s distinct vertices.

2.4 Treewidth

The definition of treewidth was first introduced in [BB73] where it was referred to as *dimension* and later rediscovered in [RS84]. First, we define the *tree decomposition* of an undirected graph $G = (V, E)$. A tree decomposition is an undirected tree $T = (V', E')$ with nodes w_1,w_2,...,w_m and a function $X\colon V' \to 2^V$, such that the following conditions hold:

1. Each vertex in V must appear in some $X(w_i)$ in T, i.e. $\bigcup_i X(w_i) = V$.

2. The subset of nodes $w_i \in V'$ for which $X(w_i)$ contains a vertex $v \in V$ must form a connected subtree in T.

3. For each pair of vertices $u, v \in V$ that are connected by an edge $(u, v) \in E$, there must exist some node w_i in T for which $u, v \in X(w_i)$.

The *width* of a tree decomposition is defined as the size of the largest subset of vertices among $\{X(w_i)\}$, minus one. The *treewidth* of a graph G is the minimal width over all possible tree decompositions of G.

3 Searching For Subgraphs

In this section we describe a polynomial time algorithm that aims to find a subgraph H in a degree bounded graph G, given that G is ϵ-far from being H-free. We start by showing a reduction of the case in which H is not connected, to the case in which H is connected. We then present a few claims and definitions relevant for general graphs. After that we show that if the graph is cycle-free and ϵ-far from being H-free, our algorithm finds some copy of H with high probability. Finally, we make a slight adjustment to the algorithm and prove that it also finds H in bounded-treewidth graphs. We denote by h the number of vertices in H. All the algorithms we present in this section are given query access to the graph G, as defined in Subsection 2.2.

3.1 Reduction of Unconnected H to Connected H

In this subsection we show that we can assume without loss of generality that H is connected. The argument is very similar to the one appearing in [CGR+12, Subsection 7.2]. Let H be a graph with connected components H_1, ..., H_m. We are interested in finding a copy of H in a graph that is ϵ-far from being H-free. Note that simply finding a copy of H_i for every $i \in [m]$, is not sufficient, since these copies might overlap or have edges between them. The algorithm presented next assumes query access to a subgraphs of G, denoted G_i. This can be achieved by omitting, from the query result on G, vertices that are not contained in G_i.

Algorithm 1. *Test-Unconnected-H-Freeness(ϵ)*

1. *Set $G_0 = G$. For $i = 1$ to m, perform*

   (a) *Invoke the H_i-freeness tester on G_{i-1} with distance parameter $\epsilon/2$. We invoke it $O(\log m)$ times, so that the probability we did not find any copy of H_i in G_{i-1}, if G_{i-1} is $\epsilon/2$-far from being H_i-free, is at most $\frac{1}{3m}$.*

(b) If the tester did not find a copy of H_i in any of its executions, then accept.

(c) Otherwise, remove from G_{i-1} the copy of H_i found by the tester. Also, remove any vertex adjacent to a vertex that belongs to the copy of H_i found. Set G_i as the resulting graph.

2. If every iteration found a copy of H_i, reject the graph.

It is easy to see that if the algorithm rejects the graph, then it found a distinct copy of H_i for every $i \in [m]$, such that there are no edges between these copies. Therefore, it found a copy of H in G. Also, the query complexity of the algorithm is $O(\log m) \cdot \sum_{i=1}^{m} q_i(\epsilon/2, k, d)$ where q_i is the query complexity of a tester for H_i-freeness.

Note that if G is ϵ-far from being H-free, then it is also ϵ-far from being H_i-free for every $i \in [m]$. Since we remove in all the iterations at most hd^2 edges, G_{i-1} is $\epsilon/2$-far from being H_i free for every $i \in [m]$ (assuming that $\frac{\epsilon dn}{2} \geq hd^2$, otherwise one can easily query the entire graph with query complexity polynomial in h, d and $1/\epsilon$). Therefore, with probability at least $1 - \frac{1}{3m}$, the i^{th} iteration will find a copy of H_i in G_{i-1}. It follows that with probability at least $2/3$, Algorithm 1 will find a copy of H in G. From this point on, we assume that H is a connected graph.

3.2 The Procedure

We start by describing a procedure that we will use for finding copies of H both in cycle-free graphs and in bounded-treewidth graphs. The parameters B and ξ will be chosen in the following subsections. The parameter R is only used for the recursion and the procedure will always be called with $R = \emptyset$.

Procedure 1. *Local-Find-H(R, B, u, ξ)*

1. *Perform a DFS scan from u, without visiting any vertex in R, for at most B steps and denote the set of vertices encountered by S.*

2. *For each $v \in S$ perform at most d queries to obtain all its neighbors.*

3. *If $G(S)$ contains H, return True.*

4. *If $\xi > 0$ then for each $v \in S - \{u\}$ call Local-Find-H($R \bigcup \{v\}$, B, u, $\xi - 1$). If one of these executions returned True then return True, otherwise return False.*

By the above description, the query complexity of Procedure 1 is $O\left(B^{\xi}\right)$. Both our algorithms for cycle-free graphs and bounded-treewidth graphs will simply perform Procedure 1, $O(1/\epsilon)$ times with different values of ξ and B until the procedure finds a copy of H or terminates. Notice that in Step 1 we perform a DFS scan, however as we explain shortly, this can be any local search on the graph (e.g. BFS).

Lemma 1. *Let U be a set of vertices such that $G(U)$ is connected. If $|N(U)| = \xi$ and $G(U)$ contains a copy of H, then for any vertex $u \in U$, executing Local-Find-H($\emptyset$, $|U|$, u, $\hat{\xi}$) with $\hat{\xi} \geq \xi$ returns True with probability 1*

Proof: We prove the claim by induction on ξ. If $|N(U)| = 0$, that is, there are no neighbors of vertices in U which are outside of U, then the DFS performed reaches all vertices in U and thus a copy of H is found in Step 3. Next, consider the case that $\xi > 0$. If the DFS scan does not visit any vertices outside U, then the set S discovered during the procedure contains U entirely and thus a copy of H is found in Step 3. In such a case the algorithm returns True. Therefore, if we reached Step 4 of the procedure this must mean we reached a vertex $v \in N(U)$. This implies that $v \in S$ and after removing it from the graph, u will be inside a set U' such that $|N(U')| \leq \xi - 1$. By induction, the procedure call Local-Find-H(G', $|U|$, u, $\hat{\xi} - 1$) with v removed should return True. □

Notice that in Lemma 1 we made no use of the fact that we perform a DFS scan, and any other local search on the graph has the same effect.

3.3 General Claims and Definitions

In this subsection we present a few general claims and definitions, relevant for general graphs, and not specifically cycle-free or bounded-treewidth graphs.

Claim 2. *If G is ϵ-far from being H-free then it contains at least $\frac{\epsilon n}{h}$ vertex-disjoint copies of H*

Proof: If G has less than $\frac{\epsilon n}{h}$ vertex-disjoint copies of H, one can simply remove all the edges incident to these vertices (less then ϵdn edges) and obtain a graph which is free from H, in contradiction to G being ϵ-far from H-freeness. □

From this point we assume that G is ϵ-far from being H-free. We let H_1,H_2,...,H_t denote the $t \geq \frac{\epsilon n}{h}$ vertex-disjoint copies of H, referred to in Lemma 2. We would like to show that with high constant probability, our algorithm will find one of these copies of H. To this end, we wish to bound the number of edges leading "out" (in some sense) of these copies of H. We first introduce some definitions. The first definition will allow us to classify vertices in the graph as follows: vertices in some H_i, vertices in some $T(H_i)$ and the rest of the vertices.

Definition 2. *For each H_i we let $T(H_i) \subseteq V$ denote the subset of vertices that satisfy the following conditions:*

1. *$T(H_i)$ is disjoint from every H_j, i.e. $T(H_i) \subseteq V \setminus \bigcup_{j=1}^{t} H_j$.*

2. *There exists a path from each vertex in $T(H_i)$ to some vertex in H_i.*

3. *Any path from $u \in T(H_i)$ to $v \notin T(H_i)$ must pass through a vertex in H_i.*

Using this definition, the following claim easily follows.

Claim 3. *$T(H_i)$ and $T(H_j)$ are disjoint for any $i \neq j$.*

Proof: To verify this, assume in contradiction that there is some $u \in T(H_i) \cap T(H_j)$. By Definition 2, there is a path P' from u to H_i and $u \notin H_i$. Let us denote by v the first vertex on the path P' which belongs to either H_i or H_j. Without loss of generality, we assume that $v \in H_i$. According to our definition of $T(H_j)$, any path from u to v must go through a vertex in H_j, in contradiction to our definition of v. A similar argument holds if $v \in H_j$. □

Definition 3. *An edge $(u, v) \in E$ is called an* exit *edge from H_i if $u \in H_i$ and $v \notin H_i \cup T(H_i)$.*

Notice that for an exit edge (u, v) from H_i, it can't hold that v is in any $T(H_j)$. By our definition $v \notin T(H_i)$ and there is a path from v to H_i not going through any $H_j \neq H_i$. We illustrate Definitions 2 and 3 in Figure 3.1.

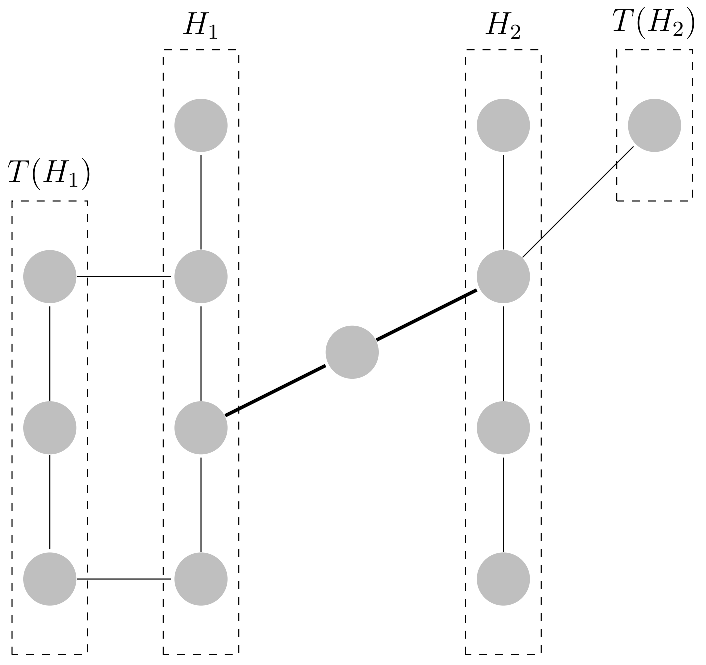

Figure 3.1: A case in which H is a k-path with $k = 4$. The dashed rectangles denote the classification of different subsets of vertices in the graph and the bold edges are the exit edges

Lemma 4. *If u belongs to H_i that has at most ξ exit edges and such that $|T(H_i)| \le \frac{4h}{\epsilon}$, then Local-Find-H($\emptyset$, $h + \frac{4h}{\epsilon}$, u, $\hat{\xi}$) with $\hat{\xi} \ge \xi$ returns True with probability 1*

Proof: Let us denote $U' \triangleq H_i \bigcup T(H_i)$. Note that $u \in U'$ and since there are at most ξ exit edges leading out of H_i, it follows that $|N(U')| \le \xi$. Therefore we can simply use Lemma 1 with $U = U'$. □

3.4 Cycle Free Graphs

In what follows, we consider the case that G is cycle-free. We start by describing the algorithm we use for finding H in cycle-free graphs that are ϵ-far from being H-free.

Algorithm 2. *Test-H-Freeness-Cycle-Free(ϵ)*

1. *Repeat the following $O(1/\epsilon)$ times.*
    (a) *Uniformly select a vertex u in G.*
    (b) *Call Procedure 1 with u, $\xi = 2$ and $B = h + \frac{4h}{\epsilon}$. If Procedure 1 returned True, the algorithm rejects.*

2. If no iteration caused rejection, accept the graph.

Our algorithm for testing H-freeness performs $O\left(\frac{h^2}{\epsilon^3}\right)$ queries. Also, it rejects the graph G only if it found some copy of H in it, so it indeed has one-sided error. We wish to establish the following theorem.

Theorem 1. *If G is cycle-free and ϵ-far from being H-free, then Algorithm 2 finds a copy of H in G with high constant probability.*

Before showing this, we need a few more claims. We now give an upper bound on the number of exit edges from a majority of the subgraphs H_j.

Lemma 5. *At least half of the sets $\{H_i\}_{i=1}^t$ have at most two exit edges*

Proof: Consider the graph G' constructed by removing from G the set of vertices which appear in $\bigcup_{j=1}^t H_j$ and $\bigcup_{j=1}^t T(H_j)$, along with the edges incident to them. We denote by $\{C_j\}_{j=1}^s$ the set of connected components in G'. Note that by the definition of connected components, there is no edge between any vertex in C_i and any vertex in C_j for $i \neq j$.

We construct an auxiliary graph G_C which has $t+s$ nodes $\{u_{H_j}\}_{j=1}^t \cup \{u_{C_j}\}_{j=1}^s$. We add an edge between u_A and u_B if there is an edge $e \in E(G)$ between some vertex in A and some vertex in B. Note that because the vertices in any set C_j are connected in G and so are the vertices in any H_j, a cycle in G_C must correspond to a cycle in G. Therefore the graph G_C is also a forest. Furthermore, there is no edge between u_{C_i} and u_{C_j} for any $i \neq j$, by the definition of connected components.

Any exit edge from H_i in G corresponds to an edge in G_C incident to u_{H_i}. Say (u, v) is some exit edge from H_i. This means that $v \notin H_i$ and v is not in any $T(H_j)$, so v must belong to some H_j or C_j. Hence, (u, v) corresponds to the edge between u_{H_i} and u_{H_j} or u_{C_j}. The opposite implication also holds. Any edge in G_C incident to some u_{H_i} corresponds to an exit edge from H_i. Therefore, the number of exit edges from H_i is equal to the number of edges in G_C incident to u_{H_i}.

Let G'_C be the graph obtained from G_C by removing all isolated nodes. This does not affect the number of edges in G_C. As we mentioned above, u_{C_i} is not connected to any u_{C_j}. So if u_{C_i} was not removed from G_C, this means it must have at least one neighbor u_{H_j}. But if it only has one neighbor, according to our definition of $T(H_j)$, it must mean that $C_j \subseteq T(H_j)$, in contradiction to the definition of C_j. It follows that any u_{C_j} remaining in G'_C must have degree of at least 2.

The average degree of nodes in a tree is less than 2. This implies that the average degree of the nodes u_{H_j} is also less than 2. Therefore at least half the nodes u_{H_j} have degree at most 2. This means that at least half the subgraphs H_j have at most 2 exit edges. $\square$

From now on we only consider copies of H that have at most two exit edges. Assume, without lost of generality, that these are the first copies of H in the union $\bigcup_i H_i$. That is, they are $H_1, H_2, ..., H_{t'}$ where $t' \geq \frac{\epsilon n}{2h}$. The number of vertices in the union of these copies of H is at least $\frac{\epsilon n}{2}$ because they are disjoint.

Lemma 6. *At least half of the sets H_1,H_2,...,$H_{t'}$ satisfy $|T(H_i)| \leq \frac{4h}{\epsilon}$*

Proof: We assume contrary to the claim that for at least half of the subgraphs it holds that $|T(H_i)| > \frac{4h}{\epsilon}$. Since there at least $\frac{\epsilon n}{2h}$ such subgraphs, and since the sets $T(H_i)$ are disjoint, we will get a contradiction because this means that there are more than $\frac{\epsilon n}{4h}\frac{4h}{\epsilon} = n$ vertices in the graph. $\square$

We are now ready to show Theorem 1.

Proof of Theorem 1: Using Lemmas 5 and 6 we now know that there are at least $\frac{\epsilon n}{4h}$ disjoint copies of H taken from the original sets H_i, for which there are only two exit edges and $T(H_i)$ is of size at most $\frac{4h}{\epsilon}$. This means that there at least $\frac{\epsilon n}{4}$ vertices that belong to the union of these sets. Using Lemma 4, we know that if we execute Procedure 1 starting from one of these vertices u with $\xi = 2$, we will surely find the copy of H which contains u. Since we perform the procedure $O(\frac{1}{\epsilon})$ times during our algorithm, each one from a randomly chosen start veretx, we have high probability of finding some copy of H. $\square$

3.5 Bounded-Treewidth Graphs

Algorithm 2 can be adapted to find, with high probability, any fixed subgraph H in G given that G is ϵ-far from being H-free and has constant treewidth w. The only modification to the algorithm is that we will apply Procedure 1 with a number of exit edges that is polynomial in w and a larger value of B (but still polynomial in ϵ, h and d).

Algorithm 3. *Test-H-Freeness-Bounded-Treewidth(ϵ, w)*

1. *Repeat the following $O(1/\epsilon)$ times.*

(a) *Uniformly select a vertex u in G.*

(b) *Call Procedure 1 with u, $\xi = 2(w+1)$ and $B = O\left(\frac{d^5 w^{O(w)} \log(d/\epsilon)}{\epsilon^3}\right)$. If the call returned True, the algorithm rejects.*

2. *If no iteration caused rejection, accept the graph.*

Algorithm 3 makes $O\left(B^{2(w+1)}/\epsilon\right)$ queries, which is polynomial in ϵ, d and h for constant values of w. We wish to show the following theorem.

Theorem 2. *If G is ϵ-far from being H-free and has treewidth w, then Algorithm 3 finds a copy of H in G with high constant probability.*

In order to show that the algorithm indeed finds a copy of H in our case, we use the following definition and claim from [EHNO11].

Definition 4 (Definition 2 in [EHNO11]). *Let $G = (V, E)$ be a graph. A subset of vertices $S' \subseteq V$ is a* neighborhood *of v in G if $v \in S'$ and the subgraph induced by S' is connected. We say that S' is an (m, δ, c)-*isolated neighborhood *of v if S' is a neighborhood of v in G, $|S'| \leq m$, $|N(S')| \leq c$ and $\frac{|N(S')|}{|S'|} \leq \delta$.*

Proposition 3 (Lemma 4 in [EHNO11]). *Let G be a graph with treewidth bounded by w. For any ϵ', δ there exists a function $g\colon V \to 2^V$ with the following properties:*

1. *For all $v \in V$, $v \in g(v)$.*

2. *$g(v)$ is connected.*

3. *There are at least $(1-\epsilon'/20)n$ vertices in G such that $g(v)$ is an $(m', \delta, 2(w+1))$-isolated neighborhood of v, with $m' = \frac{28860 d^3 (w+1)^5}{\delta \epsilon'^2}$.*

As mentioned before, Edelman et al. [EHNO11] designed a partition oracle whose complexity is polynomial in d and ϵ for the class of bounded treewidth graphs (assuming constant treewidth w). The oracle is based on a global partitioning algorithm, which accesses the whole graph, and the oracle emulates this algorithm locally. We show that if the global partitioning algorithm finds a partition with at most ϵn edges between parts, then this partition has a certain property that ensures that our algorithm finds a copy of H with high probability. Since the global partitioning algorithm presented in [EHNO11] has positive success probability (when the graph has treewidth bounded by w), such a partition must exist. This algorithm is described in the proof of the following lemma.

Lemma 7. *If G has treewidth w, then there exists a partition P of G with the following properties.*

1. *There are at most $\epsilon n/4$ edges between different parts in P.*

2. *There exists a set of vertices U such that $|U| \geq (1 - \epsilon/80)n$. Each vertex in U belongs to a part P_i for which there exists a set $S' \subseteq V$ with $|N(S')| \leq 2(w+1)$, $|S'| \leq m$ and $P_i \subseteq S'$. m is of size $O(\frac{d^5 w^{O(w)} \log(d/\epsilon)}{\epsilon^3})$.*

Proof: Algorithm 3, described in [EHNO11], attempts to construct a global partition of a graph G. The algorithm is given as input the treewidth w and the parameter ϵ'. In the proof of Theorem 1 in [EHNO11] it is shown that the parts in this partition are relatively small and that if G has treewidth w, then the algorithm has high constant probability of outputting a partition such that the total number of edges between parts is at most $\epsilon' n$.

The algorithm works by iterating over the vertices in a random order and building the different parts while doing so. For each vertex v, if v is not already contained in some part of the partition, then the algorithm attempts to find an $(m, \delta, 2(w+1))$-isolated neighborhood of v (using a deterministic algorithm similar to our Procedure 1), which we denote by S'. If v does not have such a neighborhood, S' is set to $\{v\}$. The algorithm then extracts, as a new part, the subset of vertices $\widetilde{S} \subseteq S'$ that are not already in some part in the partition. Note that $\widetilde{S}$ might contain several connected components, in that case each of these components will be a different part in the partition. The parameters m and δ are chosen so that Proposition 3 applies, implying that almost all vertices (except for an $O(\epsilon')$ fraction of them) have an $(m, \delta, 2(w+1))$-isolated neighborhood. Specifically, $m = O\left(\frac{d^5 w^{O(w)} \log(d/\epsilon')}{\epsilon'^3}\right)$ and $\delta = \frac{\epsilon'}{100 \cdot (2w+3)! \cdot (1 + \log m + \log(2w+3))}$. We set $\epsilon' = \epsilon/4$, so that there must exist some partition P, produced by the algorithm, with at most $\epsilon n/4$ edges between parts. From this point on, for the sake of succinctness, we use the shorthand *isolated neighborhood* when referring to an $(m, \delta, 2(w+1))$-isolated neighborhood.

As we described above, a new part is built in the algorithm by taking some isolated neighborhood of v (if such a neighborhood exists), denoted S', and extracting the subset of vertices in S' that are not already contained in some part previously constructed. The subset of extracted vertices constitutes the new part in the partition (there might be more than one connected component in this subset, in that case each component defines a different part). Note that, by definition, S' is also an isolated neighborhood of any other vertex in S'. Therefore, for any $v \in V$ that has an isolated neighborhood in G, the part that v belongs to in P is a subset of some isolated neighborhood that v belongs to. The parameters in the algorithm are chosen so that $|S'| = O\left(\frac{d^5 w^{O(w)} \log(d/\epsilon)}{\epsilon^3}\right)$. Since S' is an $(m, \delta, 2(w+1))$-isolated neighborhood, by definition $|N(S')| \leq 2(w+1)$. Furthermore, the parameters are chosen such that Proposition 3 applies, so we know there are at most $\epsilon n/80$ vertices that do not have such an isolated neighborhood (actually, even less than that because of details in the proof appearing in [EHNO11]). □

We are now ready to prove Theorem 2.

Proof of Theorem 2: By Lemma 7, there exists a partition P of G with the two properties stated in the lemma. For the sake of the analysis, consider removing all edges between parts and all edges incident to vertices in $V \setminus U$. In this process we remove at most $\epsilon dn/2$ edges from the graph, so the resulting graph G' is at least $\epsilon/2$-far from being H-free. Using Claim 2, at least $\epsilon n/2$ vertices in G' belong to some copy of H. We denote this set of vertices by U'. Since there are no edges incident to vertices in $V \setminus U$, all these vertices also belong to U, i.e. $U' \subseteq U$. By the definition of U, every vertex in U' belongs to a part which is a subset of some set S' of size at most m and it holds that $|N(S')| \leq 2(w+1)$. There are no edges between parts in G', so every copy of H must be completely contained in some part of the partition and therefore also contained in some S'. Using Lemma 1, we know that if we perform our procedure on G starting at any vertex in U', with $\xi = 2(w+1)$ and $B = m$, we would find that copy of H. After executing Procedure 1 $O(1/\epsilon)$ times, we find a copy of H with high probability. □

4 Random DFS Walks for Finding k-Paths

In this section we focus on the case in which H is a k-path. Our original goal was to design and analyze an algorithm whose query complexity is $\text{poly}(k, d, 1/\epsilon)$, which works for all graphs with degree bound d. We were not able to accomplish this goal, but we have made some progress, presented in this section.

We start by presenting a property of a DFS search when it is executed to find k-paths. We then present an algorithm that searches for k-paths, based on random DFS walks. After that we provide a framework for a proof that this algorithm indeed finds k-paths with high probability in graphs which are ϵ-far from being k-path free, conditioned on a conjecture regarding random DFS walks. We establish this conjecture for the specific case of cycle-free graphs. This gives us another algorithm for finding k-paths in cycle-free graphs which are ϵ-far from being k-path free, and also servers as a sanity check for the viability of the conjecture.

Finally, we prove the necessity of randomness in the DFS walks. We show this by constructing a graph that is ϵ-far from being k-path free, such that performing specific non-random DFS walks on it requires relatively many steps for finding a k-path.

4.1 Small Cuts Property

Recall that a vertex cut is a set of vertices whose removal from G renders G disconnected. Our starting point is a property of the DFS search that states that if such a search did not find any k-path, then there is a relatively small vertex cut S' separating the vertices visited during the DFS, S, from the rest of the graph. In other words, the vertex cut S' is a set of vertices whose removal from G renders the set of vertices $S \setminus S'$ disconnected from the rest of the graph. The following lemma describes this property.

Lemma 8. *If a DFS search on a connected bounded degree graph $G = (V, E)$ which visited a set of vertices S did not find a k-path, then there exists a subset $S' \subseteq S$ such that $N(S \setminus S') \subseteq S'$ and $|S'| < k$.*

Proof: The DFS holds a stack of vertices S' that were visited during the search but were not backtracked from. Every vertex $v \in S \setminus S'$ was visited during the DFS search and backtracked from. Therefore, all of the edges incident to v were traversed. This means that every vertex in $S \setminus S'$ only has edges connecting it to S, implying that $N(S \setminus S') \subseteq S'$. The set S' contains the vertices in the DFS stack, that were not backtracked from. Because the graph is connected, these vertices are connected by a path, following the order in which they were visited by the DFS. Since the DFS did not discover a k-path, it follows that $|S'| < k$. $\square$

In other words, if the DFS search failed to find a k-path while visiting s vertices, then it found a set $\widetilde{S} = S \setminus S'$ such that $|\widetilde{S}| > s - k$ and $|N(\widetilde{S})| < k$. Note that $\widetilde{S}$ might not be connected. Also, since the graph is degree bounded by d, there are at most dk edges connecting vertices in S with vertices in $V \setminus S$.

4.2 The Algorithm

We wish to use Lemma 8 in order to define a sufficient condition for proving that a random DFS based algorithm finds evidence of a k-path with high probability in a graph which is ϵ-far from being k-path free.

Algorithm 4. *Test-k-Path-Freeness(ϵ, k, d, s, α)*

1. *Uniformly select a set of $q = \frac{128s}{\epsilon}$ vertices in G, denoted Y.*
2. *For each vertex in Y, perform $p = \frac{s}{k\alpha}$ random DFS walks, each visiting a total of s vertices. If the subgraph induced by the vertices visited in any of these walks contains a k-path, reject the graph. Otherwise, accept it.*

The query complexity of the algorithm is $O\left(\frac{s^3}{\epsilon\alpha k}\right)$, where α and s are two parameters of the algorithm. When executing the algorithm, the parameter s will be chosen to be polynomial in ϵ, k and d and α will be a constant. We will also sometimes use

$p(\epsilon, k, d)$ and $q(\epsilon, k, d)$ as both of these are functions of s. Notice that as before, since the algorithm has one-sided error, we only need to show that with high probability it will find evidence of a k-path in a graph that is ϵ-far from being k-path free. First, we give a definition for an "easy" vertex and another definition which we will also use shortly.

Definition 5. *Let v be a vertex in a graph G. We will say that v is an* easy *vertex, if a random DFS starting from v and visiting a total of $s(\epsilon, k, d)$ vertices finds a k-path with probability higher than $1/p(\epsilon, k, d)$*

Definition 6. *Let v be a vertex in a graph G. We denote by $DFS^s(v)$ the random variable whose value is the set of vertices visited in a random DFS, for s steps, starting from v. We will make a slight abuse of notation and also use it as a function of a randomly chosen vertex, i.e. $DFS^s(u)$ where u is chosen randomly from the set of vertices of a graph.*

We denote by U the subset of vertices in G that are easy. Notice that if $|U| \geq c|V|/q(\epsilon, k, d)$, for a constant c, then with high probability our algorithm will sample a vertex $v \in U$. Thus, by the definition of U, one of the p walks performed starting at v will find a k-path with high probability. In this case our algorithm will find evidence of such a path with high probability. Hence we only need to handle the case in which $|U| < \frac{c|V|}{q(\epsilon,k,d)} \leq \frac{\epsilon n}{16s}$, for a large enough constant c. We now show a lemma that holds if this is indeed the case.

Lemma 9. *Suppose that the following conditions hold for a graph G:*

1. *Algorithm 4 finds a copy of a k-path with probability less then $2/3$.*
2. *There exists a set $W = \{u_1, u_2, ..., u_w\}$ of vertices in G, such that $|W| = w = \frac{\epsilon n}{8k}$ and with probability at least α, it holds that $|\bigcup_{i=1}^{w} DFS^s(u_i)| \geq \epsilon n/16$.*

Then G is ϵ-close to being k-path free.

Proof: We assume that the conditions described in the lemma hold. Let $W = \{u_1, u_2, ..., u_w\}$ be as defined in Item 2 of the lemma and observe that the random variables $DFS^s(u_i)$ are independent. Suppose we perform a single random DFS walk $DFS^s(u_i)$ from each vertex $u_i \in W$. We assume, without lost of generality, that the first w' vertices $u_1, u_2, ..., u_{w'}$ in the set W are non-easy vertices. By Definition

5, a DFS starting at a non-easy vertex has probability at most $1/p$ of finding a k-path. Therefore, the expected number of DFS walks, starting from non-easy vertices, which contain a k-path, is bounded by $w'/p \le \frac{\epsilon n}{8kp}$. Using Markov's inequality, the probability that there will be more then $\frac{\epsilon n}{4k\alpha p}$ such DFS walks is bounded by $\alpha/2$. In other words, with probability at least $1-\alpha/2$ there will be at most $\frac{\epsilon n}{4k\alpha p}$ DFS walks in $\{DFS^s(u_1), DFS^s(u_2), ..., DFS^s(u_{w'})\}$, which contain at least one k-path.

Since the probability that $|\bigcup_{i=1}^{w} DFS^s(u_i)| \ge \epsilon n/16$ is at least α, there must be at least one set of DFS walks that both cover almost the entire graph and have at most $\frac{\epsilon n}{4k\alpha p}$ walks, starting from non-easy vertices, which contain at least one k-path. We will now look at one of these sets of DFS walks and show how we use it to make the graph k-path free by removing less than ϵdn edges. We denote this fixed set of DFS walks by $S(u_i) \overset{\text{def}}{=} DFS^s_{\vec{r}_i}(u_i)$, where $\vec{r}_i$ is the vector of choices indicating which edge to traverse in each step of the DFS walk. Note that $S(u_i)$ is not a random variable. It holds that $|\bigcup_{i=1}^{w} S(u_i)| \ge \epsilon n/16$ and at most $\frac{\epsilon n}{4k\alpha p}$ of the sets $S(u_1), S(u_2), ..., S(u_{w'})$ induce a subgraph in G which contains a k-path.

For the sake of the analysis, consider removing all edges incident to vertices not in $\bigcup_{i=1}^{w} S(u_i)$. Since there are at most $\epsilon n/16$ such vertices, we will need to remove at most $\epsilon dn/16$ edges. Next, for each DFS walk $S(u_j)$ which contains a k-path, we remove all edges incident to vertices in $S(u_j)$. Since $|S(u_i)| = s$ for all i, we will need to remove at most sd edges for each $S(u_j)$ that contains at least one k-path. There are at most $\frac{\epsilon n}{4k\alpha p}$ sets $S(u_j)$, starting from a non-easy vertex u_j, which contain a k-path. Also, in the entire graph there are at most $\frac{\epsilon n}{16s}$ easy vertices. Therefore, we will remove at most $\frac{\epsilon dns}{4k\alpha p} + \frac{\epsilon dn}{16}$ edges. By choosing $1/p \le \frac{k\alpha}{s}$ we get that we need to remove at most $5\epsilon dn/16$ edges.

Now, we are only left with edges incident to vertices that belong to at least one $S(u_i)$, such that $S(u_i)$ does not contain any k-path. We denote these DFS walks by $Z = \{S(u_{i_1}), S(u_{i_2}), ..., S(u_{i_\ell})\}$. Using Lemma 8, we know that each $S(u_i)$ defines an edge-cut with at most kd edges, which separates $S(u_i)$ from the rest of the graph. Every DFS walk $S(u_j) \in Z$ does not contain any k-path, so after removing the edge cut separating $S(u_j)$ from the rest of the graph, for each $S(u_j) \in Z$, the resulting graph will be k-path free. Since $|Z| \le \frac{\epsilon n}{8k}$, we need to remove at most $\epsilon dn/8$ more edges to have the graph completely k-path free.

We made the graph k-path free by removing less than ϵdn edges, hence it is closer than ϵ to being k-path free. $\square$

We now present a conjecture regarding random DFS walks which, if proven to be correct, is enough to establish that Algorithm 4 finds a k-path in any graph that is ϵ-far from being k-path free, given the correct choice of s (polynomial in ϵ, k and d) and α (constant). There are also alternative possible conjectures to this that achieve the same result.

Conjecture 10. *For any connected graph G that is ϵ-far from being k-path free and vertex $v \in V$, if we uniformly choose a start vertex u, then $\Pr\left(v \in DFS^{s'}(u)\right) \geq \frac{1}{m(1/\epsilon,k,d)}\frac{s'}{n}$ where $m(1/\epsilon, k, d)$ is some polynomial.*

The following lemma shows that if this conjecture is indeed true, there exist some set of vertices as the one required for Lemma 9. Hence, Algorithm 4 works for general graphs.

Lemma 11. *Assume that Conjecture 10 holds. If we select a multi-set of vertices $W = \{u_1, u_2, ..., u_t\}$ uniformly at random for $t = \frac{\epsilon n}{8k}$, then $|\bigcup_{i=1}^{t} DFS^s(u_i)| \geq \epsilon n/16$ with high constant probability, for $s = \text{poly}(\epsilon, k, m)$.*

Proof: If Conjecture 10 is true, then for each $v \in V$ it holds that $\Pr\left(v \in DFS^{s'}(u)\right) \geq \frac{s'}{mn}$ for a uniformly chosen u. Therefore, it holds that

$$\Pr\left(v \notin \bigcup_{i=1}^{t} DFS(u_i)\right) = \prod_{i=1}^{t} \Pr\left(v \notin DFS(u_i)\right) \leq \left(1 - \frac{s'}{mn}\right)^{\frac{\epsilon n}{2k}} \leq e^{-\frac{\epsilon s'}{128km}}$$

By choosing $s' = s \geq 128km\frac{1}{\epsilon}\log\frac{64}{\epsilon}$ we get that the expected number of vertices not covered by these walks is at most $\epsilon n/64$ and using Markov's inequality we know that with probability at least $3/4$ we cover the entire graph except for at most $\epsilon n/16$ vertices. □

4.3 Cycle Free Graphs

We now prove Conjecture 10 for cycle-free graphs and thus show that by executing Algorithm 4 with correct parameters, one can find a k-path, with high constant probability, in every cycle-free graph which is ϵ-far from being k-path free.

Lemma 12. *Let G be a connected cycle-free graph. For any vertex $v \in V$, if we randomly choose a start vertex u and perform a random DFS walk for s steps starting at u, the probability of visiting v during the DFS is at least $\frac{s}{2d^2n}$.*

Proof: Until this point, when we said that a DFS performs s steps, we meant that it visits s distinct vertices. In this proof when we use the term DFS step we refer to a transition made by the DFS from one vertex to another, including the steps performed while backtracking from a vertex. Therefore, if a DFS performs s such steps, it might visit less than s distinct vertices, but this only helps our argument.

Let us denote the sequence of vertices encountered during the random DFS walk by $\overrightarrow{DFS}(u) = \{X_0(u), X_1(u), ..., X_s(u)\}$ where $X_i(u)$ is the vertex the DFS search encounters in its i^{th}-step. In particular, $X_0 = u$. We wish to lower bound the probability of visiting v during the DFS walk, where the probability is taken over the uniformly selected starting vertex $u \in V$ and the vector of choices indicating which edge to traverse in each step of the DFS walk, which we denote by $\vec{r}$. i.e, we seek a lower bound on $\Pr_{\vec{r},u}\left[v \in \overrightarrow{DFS}(u)\right]$.

Notice that $\overrightarrow{DFS}(u)$ is a function of a random variable - the starting vertex u, however we will make a slight abuse of notation and also use it as a function of a constant vertex (although it is still a random variable). There might be duplicates in $\overrightarrow{DFS}(u)$ because the DFS backtracks to already visited vertices during the search. We will also denote by $\overrightarrow{DFS}_i^j(u)$ the sub-sequence of vertices $\{X_i(u), X_i + 1(u), ..., X_j(u)\}$. Notice that:

$$\begin{aligned}
\Pr_{\vec{r},u}\left[v \in \overrightarrow{DFS}(u)\right] &\geq \frac{1}{n} \sum_{u' \in V \setminus \{v\}} \Pr_{\vec{r}}\left[v \in \overrightarrow{DFS}(u')\right] \\
&= \frac{1}{n} \sum_{u' \in V \setminus \{v\}} \Pr_{\vec{r}}\left[(v = X_0(u')) \cup (v = X_1(u')) \cup ... \cup (v = X_s(u'))\right] \\
&= \frac{1}{n} \sum_{u' \in V \setminus \{v\}} \sum_{1 \leq i \leq s} \Pr_{\vec{r}}\left[(v = X_i(u')) \cap (v \notin \overrightarrow{DFS}_0^{i-1}(u'))\right] \quad (4.1)
\end{aligned}$$

We need to bound the probability that a random DFS reaches v in the i^{th}-step and that it hasn't reached v before that. Similarly to the proof of Claim 6 in [Rez11], we can build a one-to-one mapping between DFS walks starting at w and reaching v for the first time in the i^{th}-step, to DFS walks starting from v and reaching w in the i^{th}-step, without returning to v before that. Let us denote by D some DFS starting from w and reaching v for the first time in the i^{th}-step. Since the graph is cycle-free, there is only one unique path connecting w to v. We denote this path by $P = \langle w = z_1, ..., z_q = v\rangle$. The graph is cycle free, therefore any DFS from w reaching

v must traverse the edges of P while occasionally leaving the path. The DFS will not stray from v, as D is defined as a DFS walk first reaching v in the last step. An example of such a DFS is depicted in Figure 4.1.

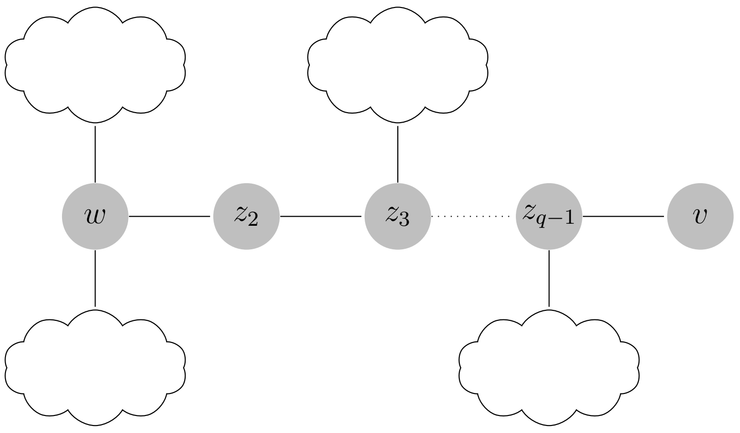

Figure 4.1: An example of a DFS walk from w to v. The clouds indicate a set of vertices to which the DFS strayed. Once it finishes visiting every vertex in the clouds, it backtracks and returns to traversing P.

We construct the bijection $M(D)$ of D by performing a DFS walk on the same vertices as D, but starting from v and reaching w at the end, i.e traversing the edges of P in the opposite direction. All other DFS steps will be the same in $M(D)$ as they were in D, except that we will first take stray walks from z_{q-1} (since we do not stray from v), then from z_{q-2} and so on until we reach w, where again we will stray just like the DFS walk D has. It is easy to see that $M(D)$ is also an i-step DFS walk and that it does not return to v before reaching w.

Almost all corresponding steps taken by D and $M(D)$ have the same probability, except for the last step taken in D (reaching v), which has no corresponding step in $M(D)$, and the first step taken in $M(D)$ (leaving v). Therefore , it holds that $\frac{1}{d}\Pr_{\vec{r}}(M(D)) \leq \Pr_{\vec{r}}(D) \leq d\Pr_{\vec{r}}(M(D))$. Using this and summing over all such DFS walks D, it is easy to see that for any $w \in V \setminus \{v\}$ it holds that:

$$\begin{aligned}\frac{1}{d}\Pr_{\vec{r}}\left[w = X_i(v), v \notin \overrightarrow{DFS}_1^{i-1}(v)\right] &\leq \Pr_{\vec{r}}\left[v = X_i(w), v \notin \overrightarrow{DFS}_0^{i-1}(w)\right] \\ &\leq d \cdot \Pr_{\vec{r}}\left[w = X_i(v), v \notin \overrightarrow{DFS}_1^{i-1}(v)\right]\end{aligned}$$

After substituting this in Equation (4.1) we get:

$$\begin{aligned}\Pr_{\vec{r},u}\left[v \in \overrightarrow{DFS}(u)\right] &\geq \frac{1}{2dn} \sum_{u' \in V \setminus \{v\}} \sum_{1 \leq i \leq s} \Pr_{\vec{r}}\left[u' = X_i(v), v \notin \overrightarrow{DFS}_1^{i-1}(v)\right] \\ &= \frac{1}{2dn} \sum_{1 \leq i \leq s} \Pr_{\vec{r}}\left[v \notin \overrightarrow{DFS}_1^{i}(v)\right]\end{aligned}$$

Since we assume the graph is connected and it's maximum degree is d, there must be an edge connecting v to a vertex $\tilde{u}$, such that if we remove all edges incident to v from G, $\tilde{u}$ will belong to a connected set of at least $\frac{n-1}{d}$ vertices. Assuming $s < \frac{n-1}{d}$, if the DFS would choose this edge as the first edge to traverse, we know that with probability 1 the DFS will not return to v in less than s steps. Therefore $\Pr_{\vec{r},u}\left[v \in \overrightarrow{DFS}(u)\right] \geq \frac{s}{2d^2n}$ □

4.4 A Lower Bound For Deterministic DFS Walks

Algorithm 4, as well as one of the algorithms presented in [Rez11], is based on selecting, uniformly at random, a set Y of vertices and for each $u \in Y$ performing a random DFS starting at u. If any of these walks finds a k-path then the graph is rejected, otherwise it is accepted. The property we presented in the previous subsection holds for any DFS performed, and not necessarily a random DFS. We were interested in the question of whether it is necessary that the DFS be random. In this subsection, we present a specific cycle-free graph, which is ϵ-far from being k-path free. We then show an ordering over the edges, such that if a deterministic DFS (following the edge order) is executed starting at any vertex in the graph, the number of steps performed before finding a k-path will be exponential in k.

We assume that k is an odd number, however this is only for our convenience and can easily be generalized. We also require that $d(k+1) < 1/\epsilon$. We construct the graph in the following manner:

1. Let T be a full $d/2$-tree with depth $k/4$ and denote its root by r.

2. For each vertex $v \in T$, add $k-1$ new vertices $P(v) = \{v_1, v_2, ..., v_{k-1}\}$. Also, add edges such that the vertices in $P(v) \cup \{v\}$ form a k-path, with v at the center. That is, connect the path $\langle v_1, ..., v_{\frac{k-1}{2}}, v, v_{\frac{k+1}{2}}, ..., v_{k-1}\rangle$.

3. Construct a separate, full $d/4$-tree with depth $k/8$, denoted T', with it's root also at r. Hence, every vertex in T' is new, except for the root r.

This graph is depicted in Figure 4.2.

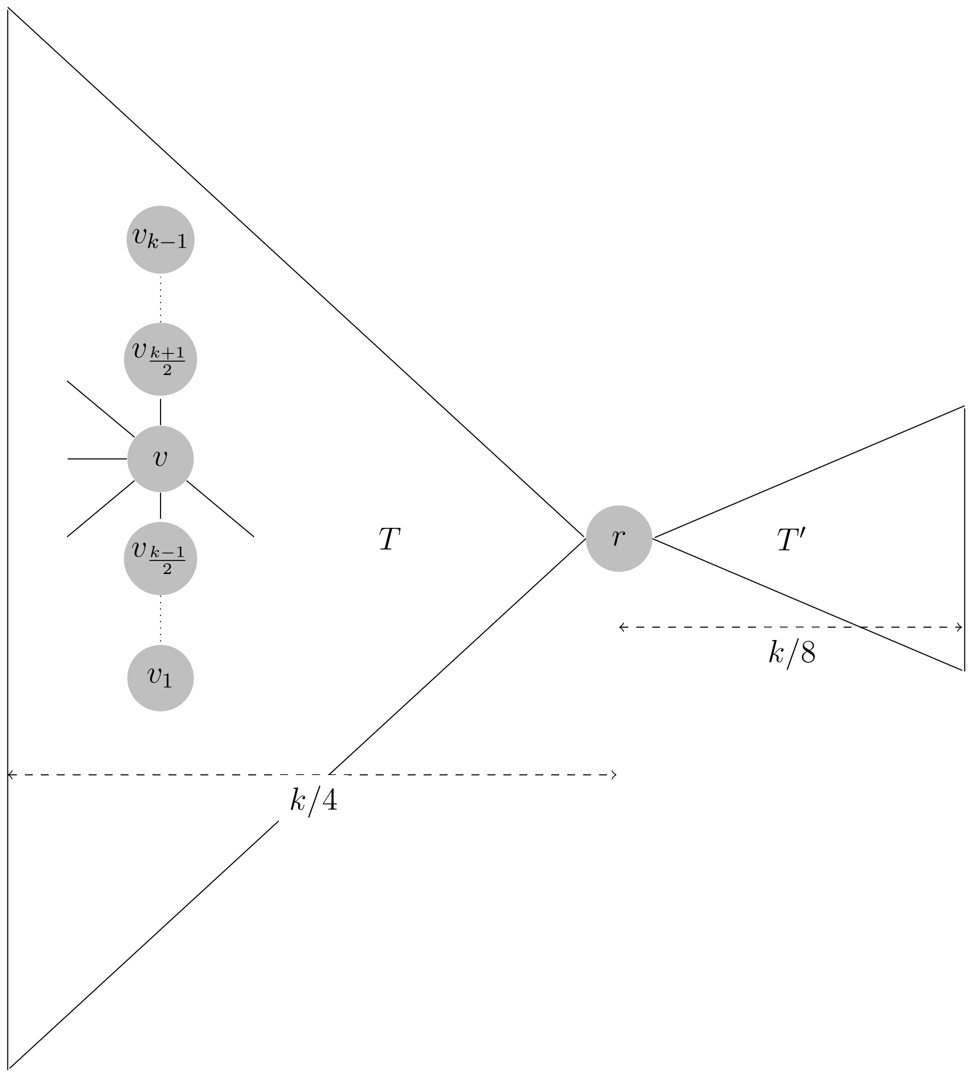

Figure 4.2: The graph we construct for showing a lower bound in the non-random DFS case. v is a vertex in T, depicted with edges connecting it to other vertices in T. Also, v is the middle vertex in the k-path $\langle v_1, ..., v_{\frac{k-1}{2}}, v, v_{\frac{k+1}{2}}, ..., v_{k-1}\rangle$.

Note that every vertex in the graph has degree less then d. The total number of vertices in the graph is $n = k|T| + |T'| - 1 < (k+1)|T|$. There are $|T|$ disjoint k-paths in the graph, so one has to remove at least $|T|$ edges to make the graph k-path free. Therefore, the graph is at least $\frac{1}{d(k+1)}$-far from being k-path free. We require $d(k+1) < 1/\epsilon$, hence it is also ϵ-far from being k-path free. We now select the DFS steps performed, starting at each possible vertex in the graph:

1. For any starting vertex $v \in T$, the DFS will traverse the edges of T towards the root, until it reaches r. Then it will start traversing T', in any order. We only require that it does not leave T' until it covered T' entirely. Since T is of depth $k/4$ and T' is of depth $k/8$, we will only find a k-path after completely covering T'.

2. For each vertex $v \in T$, if the DFS starts at a vertex in $P(v)$, the search will start by walking to v, traversing at most $k/2$ edges, and then perform the DFS as if it started at v. Again, note that no k-path will be found before covering T', since the maximum length of a path found until then is $k/2+k/4+k/8 < k$.

3. For each vertex $v \in T'$, simply traverse T', and avoid leaving it until covering T' entirely.

Notice that the different DFS walks described above define a single global ordering over the edges, such that a DFS walk starting at any vertex in the graph follows this ordering. A DFS starting at any vertex in the graph will have to cover T' entirely before finding a k-path, hence it will perform $\Omega\left((d/4)^{k/4}\right)$ steps before finding a k-path.

5 Further Observations

In this section we present a few additional approaches and ideas that we have encountered during our research. In the appendix we discuss another direction for proving our conjecture, which involves some technical calculations.

5.1 Extending the Bijection for General Graphs

Notice that in the proof of Lemma 12, the only use we made of the fact that the graph is cycle free was in the symmetric bijection between DFS walks from u to v (reaching v for the first time in the last step) and DFS walks from v to u (which do not return to v before reaching u). A natural approach would be to try and show that such a bijection, or a similar one, exists for general graphs. The problem with simply applying the original proof for general graphs is that a certain DFS walk between u and v might not even be reversible as described in the proof of Lemma 12. Consider the case depicted in Figure 5.1.

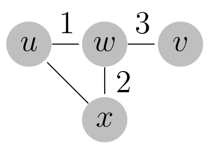

Figure 5.1: DFS walk D, starting from u and reaching v. The edge numbering indicates the order in which the edges were taken during the walk. Notice that there may be other edges incident to the depicted vertices (u, v, w, x). Also, the edge connecting x and u was not traveresed during the walk, since when we reached x we had already visited u.

The bijection we constructed for cycle-free graphs would attempt to map D to the (illegal) DFS walk depicted in Figure 5.2.

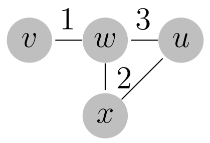

Figure 5.2: $M(D)$, starting from v and reaching u. The edge numbering indicates the order in which the edges were taken during the walk. This is not a legal DFS walk, since when we reached x we were not allowed to backtrack from it to w, because it has an incident edge leading to u, and u was uncovered at that point.

5.1.1 Building a Bijection Between Sets of DFS Walks

An alternative approach one might take, would be to construct a bijection between sets of DFS walks leading from u to v and sets of DFS walks leading from v to u. Consider a DFS walk D in a connected graph G, leading from u to v, reaching v in its last step. The DFS holds a stack of vertices, connected by a path, that were visited during the DFS but were not backtracked from. We denote these vertices by $P(D) = (z_1, ..., z_q)$ where $z_1 = u$ and $z_q = v$. We also denote by $P^R(D)$ the reversed path, i.e., $P^R(D) = (z_q, ..., z_1)$. The walk D traverses the edges connecting (z_i, z_{i+1}) for $1 \leq i \leq q-1$ while occasionally straying from the path P, and returning to it afterwards. In each such excursion, an entire connected component (if we were to remove P from the graph) was completely traversed and backtracked from. Let us denote these components by $C(D) = \{C_1, ..., C_t\}$.

Consider some components C_i. There is some vertex $z_j \in P$, such that when the walk D visited z_j, it selected the edge connecting z_j to a vertex in C_i. However, note that there might also exist an edge between C_i and some vertex from the set $\{z_1, ..., z_{j-1}\}$, which is what made our original bijection invalid for the general case.

We define $DFS(u, v, P, C)$ as the set of all DFS walks D from u to v, reaching v for the first time in the last step, such that $P(D) = P$ and $C(D) = C$. We also define $DFS^R(u, v, P, C)$ as the set of all DFS walks D' from v to u, which do not return to v before reaching u, such that $P(D') = P^R$ and $C(D') = C$. Our bijection will map the set $DFS(u, v, P, C)$ to the set $DFS^R(u, v, P, C)$.

Given the set $DFS(u, v, P, C)$, one can construct $DFS^R(u, v, P, C)$ by first reversing P and getting $P^R = (z_q, ..., z_1)$. Then, the excursion to each $C_i \in C$ must take place from the minimal j such that z_j is connected by an edge to some vertex in C_i, since otherwise the DFS will not be legal. We denote by $\alpha(z_j)$ the set of edges leading from z_j to some component in C, such that there is no edge from z_k with $k < j$ to that component in C. The set $DFS^R(u, v, P, C)$ will contain every permutation possible over the edges in $\alpha(z_j)$, for every $1 \leq j \leq q$. Each permutation indicates the order in which the edges leading out of z_k into sets in C are taken (there might be multiple edges leading from z_j to some C_i, the DFS will only take the first of them occurring in the permutation). An example of the mapping is depicted in Figure 5.3.

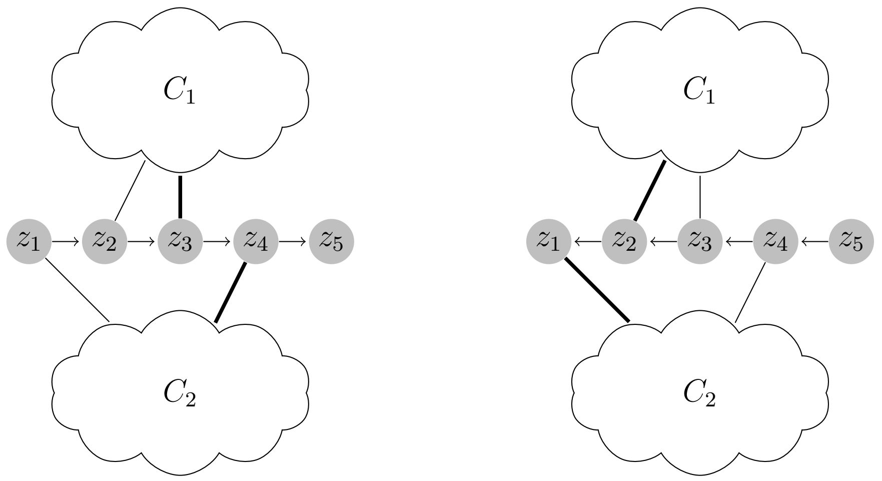

Figure 5.3: An example of a mapping between a single DFS from z_1 to z_5 and a single DFS from z_5 to z_1. The bold edges indicate edges actually taken by the DFS. Note that if, for example, there were more edges connecting z_4 and C_2, there would have been more possible DFS walks in the left figure.

This bijection is indeed valid. However, it does not hold the main property we need - that the sum over all probabilities of DFS walks in $DFS(u, v, P, C)$ is close (up to a polynomial factor) to the sum over all probabilities of DFS walks in $DFS^R(u, v, P, C)$. Consider the case in which $P = (z_1, ..., z_{k-1})$, $C = \{C_1, ..., C_{d-2}\}$ such that there are $d-2$ additional edges incident to z_{k-2}, connecting it to a vertex in each C_i. Also, for each $1 \leq i \leq d-2$, there are $d-2$ edges, each connecting z_{i+1} to some vertex in C_i. This is only possible if $d+1 < k$. The set $DFS(u, v, P, C)$ contains DFS walks over P that stray from P when visiting z_{k-2}. The sum over all probabilities of DFS walks in $DFS(u, v, P, C)$ is $d^{-(d-1)}$, because we need to avoid taking any edge incident to z_i, leading out of P, for $i < k-2$, and we need to traverse all edges incident to z_{k-2}, leading out of P, in some order, before moving to z_{k-1}. On the other hand, the set $DFS^R(u, v, P, C)$ holds DFS walks, such that the excursion to C_i is taken from the vertex z_{i+1}. The sum over all probabilities of DFS walks in $DFS^R(u, v, P, C)$ is $\frac{1}{d}\left(1-\frac{1}{d}\right)^{d-2} > \frac{1}{2de}$ since we need to avoid straying from P^R when visiting z_{k-2}, and then take some edge leading to C_i when visiting z_{i+1} before moving on to z_i.

5.2 An Auxiliary Graph for Analyzing Random DFS Walks

We would have liked to better understand properties of random DFS walks. As we mentioned in the introduction, we were unable to find previous work on random DFS walks. A tool one might use, is to model the random DFS walk as a finite Markov chain $\mathcal{M} = (X_0, ..., X_k)$. A state in the state space S of this chain is a pair $(\vec{v}, I)$ where $\vec{v}$ is a vector of length $0 \leq |\vec{v}| \leq n$, with elements which are vertices in the graph (without repetitions in the vector), and $I \subseteq E$. $\vec{v}$ represents the sequence of vertices that were visited during the DFS walk and not yet backtracked from. I are edges incident to vertices in $\vec{v}$ that were already taken during the DFS walk, or leading to vertices that were already visited during the DFS. The vertex at position i in $\vec{v}$, $\vec{v}_i$, is the i^{th} vertex in the stack that the DFS keeps, which holds vertices that were not backtracked yet from.

The initial state would be $X_0 = (\vec{v}, I) = (\emptyset, \emptyset)$. The transition probability between two states $\Pr[X_i = (\vec{v}, I) \mid X_{i-1} = (\vec{v}', I')]$ would be:

1. 0 if it impossible to change from state X_{i-1} to X_i in a single step.

2. 1 if all edges incident to $\vec{v}'_{|\vec{v}'|}$ appear in I' and X_i is the next appropriate state.

3. 1 over the number of edges incident to $\vec{v}'_{|\vec{v}'|}$ which do not appear in I' otherwise

One can now use methods for analyzing Markov chains, in order to understand properties of random DFS walks. For instance, the probability of being at a vertex u during the i^{th}-step of the DFS would be $\sum_{(\vec{v},I)\in S, \vec{v}_{|\vec{v}|}=u} \Pr[X_i = (\vec{v}, I)]$. Note that we are interested in walks that find k-paths, so it might be helpful to limit the length of the vector $\vec{v}$ to $k-1$, since otherwise we already found a k-path.

5.3 Partition Oracle for Graphs Without k-Paths

As we discussed in the introduction, there has been work done on designing partition oracles for various families of graphs, and in particular, graphs with an excluded minor. We noted in Subsection 1.4.1 that a quasi-polynomial partition oracle was given for graphs with an excluded minor in [LR13]. It is still unknown whether there exists a partition oracle for graphs with an excluded minor, which only makes a polynomial number of queries.

A problem one might consider is of designing a partition oracle for graphs without a k-path minor. As we explained in Subsection 1.6.1, this is equivalent to designing a partition oracle for graphs without a k-path as a subgraph. The question that arises is whether it is possible to build a partition oracle that makes a polynomial number of queries (assuming constant k) for this particular family of excluded minors. This is similar to work done in [EHNO11] for graphs with constant tree-width.

Recall the partition oracle given in [EHNO11], that we explained in Subsection 3.5. One approach that we considered, is making a small variation in the procedure used in [EHNO11] for finding an isolated neighborhood of a vertex v, while keeping the rest of the algorithm the same as in [EHNO11]. An isolated neighborhood of a vertex v is, intuitively, a connected subset of vertices that contains v and has a small vertex cut separating it from the rest of the graph. As we have shown in Lemma 8, if the graph G has no k-path, then the DFS we perform also finds a small vertex cut seperating almost all of the vertices visited during the DFS from the rest of the graph. So it is tempting to simply replace the procedure used in [EHNO11] with performing a DFS walk (even a deterministic one). However, the problem that arises is that when we perform a DFS walk starting at v, the subset of vertices found with a small vertex cut might not contain v. Also, it might not be connected. Thus, it is unclear how to find an isolated neighborhood of v in a graph that is k-path free.

References

[ABLS07] N. Alon, I. Benjamini, E. Lubetzky, and S. Sodin. Non-backtracking random walks mix faster. *Communications in Contemporary Mathematics*, (9):585–603, 2007.

[BB73] U. Bertel and F. Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory*, pages 137–148, 1973.

[BSS08] I. Benjamini, O. Schramm, and A. Shapira. Every minor-closed property of sparse graphs is testable. *Proceedings of the Fourtieth Annual ACM Symposium on the Theory of Computing*, pages 393–402, 2008.

[CGR$^+$12] A. Czumaj, O. Goldreich, D. Ron, C. Seshadhri, A. Shapira, and C. Sohler. Finding cycles and trees in sublinear time. *Random Structures and Algorithms*, pages 139–184, 2012.

[EHNO11] A. Edelman, A. Hassidim, H. N. Nguyen, and K. Onak. An efficient partitioning oracle for bounded-treewidth graphs. *Proceedings of RANDOM*, pages 530–541, 2011.

[GGR98] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *JACM*, pages 653–750, 1998.

[GR97] O. Goldreich and D. Ron. Property testing in bounded degree graphs. *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, pages 406–415, 1997.

[HKNO09] A. Hassidim, J. A. Kelner, H. N. Nguyen, and K. Onak. Local graph partitions for approximation and testing. *Proceedings of FOCS*, pages 22–31, 2009.

[LR13] R. Levi and D. Ron. A quasi-polynomial time partition oracle for graphs with an excluded minor. *ICALP*, pages 709–720, 2013.

[Rez11] A. Reznik. Finding k-paths in cycle-free graph. Master's thesis, Weizmann Institute of Science, 12 2011.

[RS84] N. Robertson and P. D. Seymour. Graph minors iii: Planar tree-width. *Journal of Combinatorial Theory*, pages 49–64, 1984.

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

A Iterative Approach

In Lemma 12 we've shown that in the case of cycle free graphs, if we perform a DFS walk for s steps starting from a uniformly chosen vertex, then the probability of finding any given vertex in the graph is at least $\frac{s}{p(d)n}$ where $p(d)$ is a polynomial in d. An approach one might take, for general graphs, is showing this iteratively. We know that this property holds for trees, so by showing that after adding each edge the lower bound on the probability of reaching a vertex in the graph isn't rapidly decreasing, we might hope to prove this for general graphs. We first introduce the following definition.

Definition 7. *For a vertex v in $G = (V, E)$ its neighborhood of depth s' is the set of vertices of distance at most s' from v, i.e., $N^{s'}(v) = \{u \in V \mid \operatorname{dist}(u, v) \leq s'\}$.*

For a graph $G = (V, E)$, we start by looking at the first edge added to some tree connecting all the vertices in the graph, spanned by the original edges E. In other words, given some order on E, the first edge that creates a cycle in the graph.

First, we note that the proof for cycle free graphs can be extended to a more general class of graphs. If it holds that for every vertex $v \in V$ the subgraph induced by $N^s(v)$ forms a tree, then Lemma 12 holds. This is easy to see by noticing that any DFS walk visiting a vertex not in $N^s(v)$ will never reach v in s steps. Therefore, one can obtain E' by removing from E any edge that is not in the subgraph induced by $N^s(v)$, add new edges to E' such that the entire graph becomes a tree, and still get the same probability of reaching v in a random DFS walk for s setps as it was in G.

Following this observation, we actually only need to examine the first edge added to the cycle-free graph which, for some vertex u, connects two vertices v, w for which $\operatorname{dist}(u, v) + \operatorname{dist}(u, w) < 2s$. Actually, it can be proven that only edges connecting two vertices of distance at most s from u are of interest to us.

We were not able to find an iterative proof for lower bounding the probability of reaching a vertex in G, during a DFS walk for s steps. In the rest of this section, we examine a few specific examples of cycle-free graphs that we looked at in order to better understand the effect of adding an edge to them.

A.1 Chain Graph

We start by examining the simple case of a chain graph, i.e., $G = (V, E)$ with $V = \{v_1, v_2, ..., v_n\}$ and $E = \bigcup_{i=1}^{n-1}\{(v_i, v_{i+1})\}$. We consider an edge $e = (v_j, v_k)$ added to E and we are interested in calculating the change in the probability of reaching the vertex v_i in a random DFS walk. For simplicity, we assume that i, j, k are all far enough from 1 and n (with respect to the size of s) so we don't have to take into considerations the effect of reaching either ends of the chain during the DFS walk. Before adding the edge, the probability of reaching v_i during a random DFS walk for s steps, starting at a uniformly chosen vertex, is:

$$\Pr_{\vec{r},u}\left(v_i \in DFS(u)\right) = \frac{1}{n} + \frac{2s}{2n} = \frac{s+1}{n}$$

First, we handle the case in which $i < j < k < i + s$. The probability of reaching v_i is now:

$$\begin{aligned}\Pr_{\vec{r},u}\left(v_i \in DFS(u)\right) &= \frac{1}{n} + \frac{s}{2n} + \frac{j-i-1}{2n} \\ &+ \frac{1}{3n} + \frac{k-j-1}{2n} + \frac{2}{3n} \\ &+ \frac{s-k+i}{2n} + \frac{k-j-1}{4n} = \frac{s+1}{n} + \frac{k-j-1}{4n}\end{aligned}$$

Now, for the case $i < j < i + s < k < 2s + 2i - j$:

$$\begin{aligned}\Pr_{\vec{r},u}\left(v_i \in DFS(u)\right) &= \frac{1}{n} + \frac{s}{2n} + \frac{j-i-1}{2n} \\ &+ \frac{1}{3n} + \frac{s+i-j}{4n} + \frac{s+i-j-1}{4n} \\ &+ \frac{1}{6n} + \frac{s+i-j-1}{4n} = \frac{s+1}{n} + \frac{s+i-j-2}{4n}\end{aligned}$$

Notice that this does not depend on k. In both cases, the probability of reaching v_i increased after adding an edge.

A.2 Full Tree Graph

We now examine the case of a full tree graph, where each vertex has d neighbors (except for leaves) and we assume the degree bound is $d + 1$. During this analysis, we avoid handling vertices near leaves (with respect to the size of s). For any vertex v in the tree, there are $d(d-1)^{i-1}$ vertices of distance i from v and each such vertex starting a DFS walk has probability $\frac{1}{d(d-1)^{i-1}}$ of reaching v during the walk. So the probability of reaching v during a DFS walk for s steps starting at a uniformly chosen vertex is $\frac{s+1}{n}$. In the following analysis, we denote by Δ the difference incurred in the probability of reaching v as a result of adding an edge, i.e., the probability of reaching v before we add the edge minus the probability after.

A.2.1 Adding an edge between a vertex and a descendant of it

Consider adding an edge between a vertex v_i at distance i from v and a descendant of it v_j of distance j from v. Also $i < s$ and $j < s$. This case is depicted in Figure A.1.

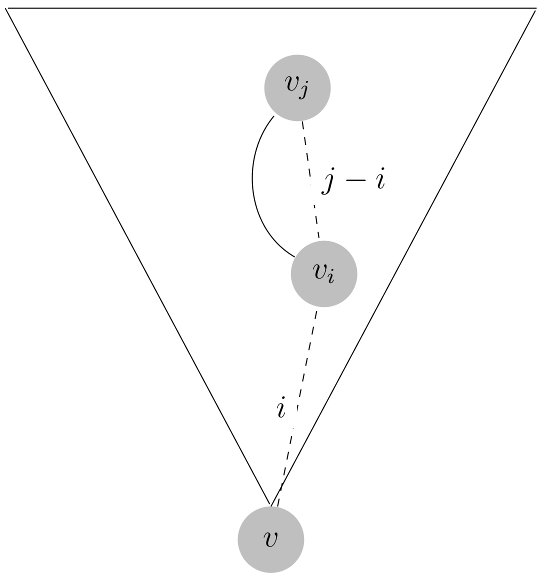

Figure A.1: Tree rooted at v, such that v_j is a descendant of v_i. We add an edge connecting v_i and v_j, depicted as the curved edge

Notice that for each vertex in the graph, there are at most two possible paths for reaching v: traversing the tree edges connecting the vertex to v, just as before, and walking to v_j, moving to v_i using the newly added edge and walking towards v using the tree edges. We sum over these paths seperately. If we look at the original tree as if it was rooted at v, every vertex that is not a descendant of v_i has the same probability of reaching v. This is true because it can only reach v using the first type of paths, and since this path does not include v_i or v_j, the probability remains the same. So we actually only need to compare the probability of reaching v before and after adding the edge from vertices in the subtree of v_i. Let us denote this subtree by T_i. Also, we will denote by v_{i+1} the vertex connected to v_i which is on the path connecting v_i and v_j.

We start by upper bounding the difference in the probability of reaching v before and after adding the edge, indicating that the probability decreases and then we will give an exact analysis of this case. Let us calculate the probability before we add the edge. We are only interested in the subtree of v_i, so we will denote by k the distance of a vertex u from v_i. The probability of reaching v after starting a DFS at u is $\frac{1}{d(d-1)^{i+k-1}}$ and there are $(d-1)^k$ vertices of distance k from v_i. Therefore, it holds that:

$$\Pr_{\vec{r},u}\left(v \in DFS(u), u \in T_i\right) = \sum_{k=0}^{s-i} \frac{(d-1)^k}{nd(d-1)^{i+k-1}} = \frac{s-i+1}{nd(d-1)^{i-1}}$$

We next examine the two types of paths, after we add the edge:

1. For the first type of paths, the probability of reaching v from any vertex in T_i decreases (if it was non zero to begin with). This is because the degree of v_i and v_j increases. Notice that if we start the DFS from a descendant of v_j, in order to remain in the first type of path we must not take the newly added edge connecting v_j to v_i when we visit v_j, but on the other hand we don't have the option of taking the edge from v_i to v_j because we already visited v_j. Similarly, if we start a DFS at a vertex in T_i which is not a descendant of v_j, we must not take the edge leading from v_i to v_j. Therefore, the probability of reaching v using the first type of paths, starting at a vertex of distance k from v_i is $\frac{1}{d^2(d-1)^{k+i-2}}$. So the probability of reaching v if we start at a vertex in T_i,

using the first type of path, is:

$$\begin{aligned}
&\sum_{k=0}^{s-i} \frac{(d-1)^k}{nd^2(d-1)^{i+k-2}} - \frac{1}{nd^2(d-1)^{i-2}} \\
&+ \frac{1}{n(d+1)(d-1)^{i-1}} - \frac{1}{nd^2(d-1)^{j-2}} + \frac{1}{n(d+1)(d-1)^{j-1}} \\
=&\frac{s-i+1}{nd^2(d-1)^{i-2}} + \frac{1}{nd^2(d+1)(d-1)^{i-1}} + \frac{1}{nd^2(d+1)(d-1)^{j-1}}
\end{aligned}$$

Where the additional terms are added because we need to give special care for v_i and v_j.

2. For the second type of paths, the analysis becomes a bit more complicated. A vertex at distance k from v_j will need to traverse $k+1+i$ edges in order to reach v using this type of paths, so we can examine all vertices of distance at most $s-i-1$ from v_j, i.e, look at the tree as if it was rooted at v_j and sum these probabilities for all vertices in the subtree composed of vertices with distance at most $s-i-1$ from v_j. However, there are certain vertices in this subtree for which this path is not possible: v_i and any vertex u for which the path connecting u to v_j passes through v_i. Additionaly, even for vertices for which this path is possible, the probability of reaching v_j is not always of the same form. For most vertices of distance $k \geq 1$ from v_j, the probability of this path is $\frac{1}{d^3(d-1)^{k+i-2}}$ because there are three vertices we visit during the walk for which the probability of choosing the correct edge is $1/d$: the start vertex, v_j and v_i. However, there is a set of vertices for which this probability is $\frac{1}{d^2(d-1)^{k+i-1}}$, because if we already visited v_{i+1} during the walk, when we reach v_i we will have probability $1/(d-1)$ of choosing the correct edge. This set of vertices are the descendants of v_{i+1} which are not descendants of v_i.

We upper bound the probability for these paths, by ignoring these two problems. We sum over all vertices of distance at most $s-i-1$ from v_j and we will take the higher probability for each of them. We get that the probability of reaching v using the second type of paths is:

$$\begin{aligned}
&\sum_{k=1}^{s-i-1} \frac{(d-1)^k}{nd^2(d-1)^{k+i-1}} + \frac{1}{n(d+1)d(d-1)^{i-1}} \\
&= \frac{s-i-1}{nd^2(d-1)^{i-1}} + \frac{1}{n(d+1)d(d-1)^{i-1}} \\
&= \frac{s-i}{nd^2(d-1)^{i-1}} - \frac{1}{nd^2(d+1)(d-1)^{i-1}}
\end{aligned}$$

Now if we take the original probability of reaching v, before adding the edge, and subtract the new probability we get:

$$\begin{aligned}
\Delta &\geq \frac{s-i+1}{nd(d-1)^{i-1}} - \frac{s-i+1}{nd^2(d-1)^{i-2}} - \frac{s-i}{nd^2(d-1)^{i-1}} - \frac{1}{nd^2(d+1)(d-1)^{j-1}} \\
&= \frac{s-i+1}{nd^2(d-1)^{i-1}} - \frac{s-i}{nd^2(d-1)^{i-1}} - \frac{1}{nd^2(d+1)(d-1)^{j-1}} \\
&= \frac{1}{nd^2(d-1)^{i-1}} - \frac{1}{nd^2(d+1)(d-1)^{j-1}}
\end{aligned}$$

It is easy to see that the probability decreases after we add the edge connecting v_i and v_j. Actually, the probability decreases by much more, because the bound we gave for the second type of paths is not tight at all.

In order to find the exact value of Δ, instead of a lower bound, we give an accurate calculation for the second type of paths. As we have said, we are interested in vertices at distance of at most $s-i-1$ from v_j. We will now look at the tree as rooted at v_j. With this outlook, we will not want to include in the sum the vertex v_i or any vertex which is a descendant of v_i, because for these vertices this path is not possible. Also, we will need to give special consideration to vertices which are descendants of v_{i+1} and not v_i, because as we have mentioned before, for these vertices the probability of the second type of paths is slightly higher. A vertex of distance k from v_{i+1} will have to traverse k edges to reach v_{i+1}, another $j-i-1$ edges to reach v_j and then $i+1$ edges for walking towards v. So we will be interested in vertices which are decendents of v_{i+1}, not v_i, and of distance at most $s-j$ from v_{i+1}. Therefore the exact calculation is now:

$$\begin{aligned}
&\sum_{k=1}^{s-i-1} \frac{(d-1)^k}{nd^3(d-1)^{k+i-2}} + \frac{1}{n(d+1)d(d-1)^{i-1}} - \sum_{k=0}^{s-j} \frac{(d-1)^k}{nd^3(d-1)^{k+j-3}} \\
&+ \sum_{k=1}^{s-j} \frac{(d-2)(d-1)^{k-1}}{nd^2(d-1)^{k+j-2}} + \frac{1}{nd^2(d-1)^{j-2}} \\
&= \frac{s-i-1}{nd^3(d-1)^{i-2}} + \frac{1}{n(d+1)d(d-1)^{i-1}} - \frac{s-j+1}{nd^3(d-1)^{j-3}} + \frac{(s-j)(d-2)}{nd^2(d-1)^{j-1}} + \frac{1}{nd^2(d-1)^{j-2}} \\
&= \frac{s-i}{nd^3(d-1)^{i-2}} + \frac{1}{n(d+1)d(d-1)^{i-1}} - \frac{s-j+1}{nd^3(d-1)^{j-3}} + \frac{s-j+1}{nd^2(d-1)^{j-2}} - \frac{s-j}{nd^2(d-1)^{j-1}} \\
&= \frac{s-i}{nd^3(d-1)^{i-2}} + \frac{1}{n(d+1)d(d-1)^{i-1}} + \frac{s-j+1}{nd^3(d-1)^{j-2}} - \frac{s-j}{nd^2(d-1)^{j-1}} \\
&= \frac{s-i}{nd^3(d-1)^{i-2}} + \frac{1}{n(d+1)d(d-1)^{i-1}} + \frac{1}{nd^3(d-1)^{j-2}} - \frac{s-j}{nd^3(d-1)^{j-1}}
\end{aligned}$$

So, again, if we subtract the new probability of reaching v, after adding the edge, from the original probability, we get:

$$\begin{aligned}
\Delta =& \frac{s-i+1}{nd(d-1)^{i-1}} - \frac{s-i+1}{nd^2(d-1)^{i-2}} - \frac{1}{nd^2(d+1)(d-1)^{i-1}} - \frac{1}{nd^2(d+1)(d-1)^{j-1}} \\
&- \frac{s-i}{nd^3(d-1)^{i-2}} - \frac{1}{n(d+1)d(d-1)^{i-1}} - \frac{1}{nd^3(d-1)^{j-2}} + \frac{s-j}{nd^3(d-1)^{j-1}} \\
=& \frac{s-i+1}{nd^2(d-1)^{i-1}} - \frac{s-i}{nd^3(d-1)^{i-2}} + \frac{s-j}{nd^3(d-1)^{j-1}} \\
&- \frac{1}{nd^2(d+1)(d-1)^{i-1}} - \frac{1}{nd^2(d+1)(d-1)^{j-1}} - \frac{1}{n(d+1)d(d-1)^{i-1}} - \frac{1}{nd^3(d-1)^{j-2}} \\
=& \frac{s-i}{nd^3(d-1)^{i-1}} + \frac{s-j}{nd^3(d-1)^{j-1}} - \frac{1}{nd^3(d-1)^{j-2}} - \frac{1}{nd^2(d+1)(d-1)^{j-1}} \qquad \square
\end{aligned}$$

It is easy to see that the probability of reaching v decreased after adding an edge connecting v_i and v_j. The expression we have gotten for Δ gave us some intuition to the effect adding an edge has on the probability of reaching v. We thought that one might be able to show that the probability of reaching v changes, as a result of adding an edge, in an amount that is proportional to the probability of reaching v during a random DFS while also visiting v_i or v_j.

A.2.2 Adding an edge between vertices with their least common ancestor at the root

We also examine the simpler case in which v_i and v_j are not descendants of one another, but instead the path connecting them passes through v, i.e, their least common ancestor in the tree rooted at v is v. We also assume $i+j \geq s$, as this is the best case in our scenario, i.e, with the highest probability of reaching v. The reason for this will become clear soon. This case is depicted in Figure A.2.

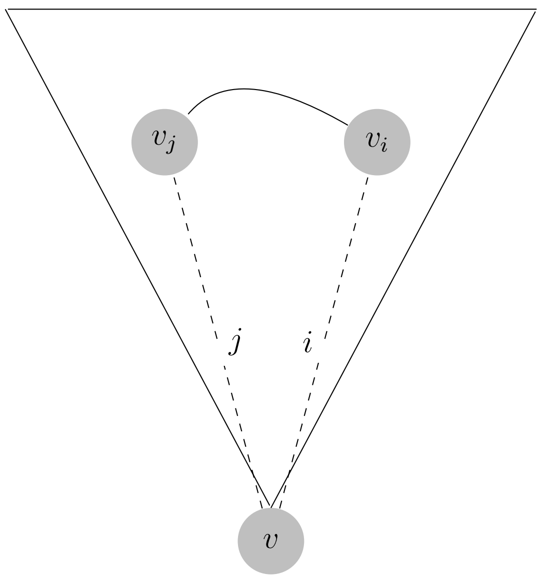

Figure A.2: Tree rooted at v, such that v_i and v_j have v as their least common ancestor. We add an edge connecting v_i and v_j, depicted as the curved edge

Similarly to the case in which v_j is a descendant of v_i, the only original paths for which the probability changes are paths from vertices which are descendants of either v_i or v_j to v. For every other vertex the same path is still available and has the same probability. Additionaly, after adding the edge, there are new DFS walks possible: starting at a vertex of distance $k \leq s-i-1$ from v_j, walking towards v_j, moving to v_i and traversing to v. Of course, the same can be said by switching i and j.

Again, we will denote descendants of v_i by T_i and descendants of v_j by T_j. Just as before, the probability of reaching v with a DFS starting at T_i or T_j before we add the edge is:

$$\Pr_{\vec{r},u}(v \in DFS(u), u \in T_i) + \Pr_{\vec{r},u}(v \in DFS(u), u \in T_j) = \frac{s-i+1}{nd(d-1)^{i-1}} + \frac{s-j+1}{nd(d-1)^{j-1}}$$

After we add the edge, the probability of the original paths is:

$$\sum_{k=1}^{s-i} \frac{(d-1)^k}{nd^2(d-1)^{i+k-2}} + \frac{1}{n(d+1)(d-1)^{i-1}} + \sum_{k=1}^{s-j} \frac{(d-1)^k}{nd^2(d-1)^{j+k-2}} + \frac{1}{n(d+1)(d-1)^{j-1}}$$
$$= \frac{s-i}{nd^2(d-1)^{i-2}} + \frac{1}{n(d+1)(d-1)^{i-1}} + \frac{s-j}{nd^2(d-1)^{j-2}} + \frac{1}{n(d+1)(d-1)^{j-1}}$$

We calculate the probability of the new paths by looking at the graph as if it was rooted at v_i (respectively, v_j). A vertex of distance k from v_i will need to traverse k edges to reach v_i, take the newly added edge to v_j and another j edges to reach v. Since $i + j \geq s$ this path is possible for any vertex of distance $k \leq s - j - 1 \leq i - 1$ from v_i and the probability of such a path is $\frac{1}{d^3(d-1)^{k+j-2}}$. Therefore the probability of the new paths is:

$$\sum_{k=1}^{s-i-1} \frac{d(d-1)^{k-1}}{nd^3(d-1)^{k+i-2}} + \frac{1}{n(d+1)d(d-1)^{i-1}} + \sum_{k=1}^{s-j-1} \frac{d(d-1)^{k-1}}{nd^3(d-1)^{k+j-2}} + \frac{1}{n(d+1)d(d-1)^{j-1}}$$
$$= \frac{s-i-1}{nd^2(d-1)^{i-1}} + \frac{1}{n(d+1)d(d-1)^{i-1}} + \frac{s-j-1}{nd^2(d-1)^{j-1}} + \frac{1}{n(d+1)d(d-1)^{j-1}}$$

We now calculate the difference in the probability of reaching v: the probability of reaching v before we add the edge minus the probability after.

$$
\begin{aligned}
\Delta =& \frac{s-i+1}{nd(d-1)^{i-1}} + \frac{s-j+1}{nd(d-1)^{j-1}} \\
&- \frac{s-i}{nd^2(d-1)^{i-2}} - \frac{1}{n(d+1)(d-1)^{i-1}} - \frac{s-j}{nd^2(d-1)^{j-2}} - \frac{1}{n(d+1)(d-1)^{j-1}} \\
&- \frac{s-i-1}{nd^2(d-1)^{i-1}} - \frac{1}{n(d+1)d(d-1)^{i-1}} - \frac{s-j-1}{nd^2(d-1)^{j-1}} - \frac{1}{n(d+1)d(d-1)^{j-1}} \\
=& \frac{s-i}{nd^2(d-1)^{i-1}} + \frac{s-j}{nd^2(d-1)^{j-1}} + \frac{1}{n(d+1)d(d-1)^{i-1}} + \frac{1}{n(d+1)d(d-1)^{j-1}} \\
&- \frac{s-i-1}{nd^2(d-1)^{i-1}} - \frac{1}{n(d+1)d(d-1)^{i-1}} - \frac{s-j-1}{nd^2(d-1)^{j-1}} - \frac{1}{n(d+1)d(d-1)^{j-1}} \\
=& \frac{1}{nd^2(d-1)^{i-1}} + \frac{1}{nd^2(d-1)^{j-1}} \quad \square
\end{aligned}
$$

Similarly to the previous case, in which the edge added connects two vertices such that one is a descendant of the other, we showed that the probability of reaching v during a random DFS decreases as a result of adding the edge connecting v_i and v_j. However, in this case the amount by which the probability decreased is smaller, and is only proportional to the probability of reaching v from v_i or v_j during a random DFS.

A.2.3 Adding an edge between vertices with their least common ancestor at the root while preserving vertex degrees

One might consider our comparsion between the tree before adding the edge and the tree after to be "unfair", since we are comparing a graph of maximal degree d with a graph of maximal degree $d+1$. So instead we will now consider a graph which is almost a full d-tree. Each vertex, which is not a leaf, will have d neighbors in the tree, except for v_i and v_j which will have $d-1$ neighbors in the tree. Also, v_i and v_j will be connected by an edge.

Just like in the previous analysis, there are two possible types of paths: paths which use only the tree edges, and paths which also use the edge connecting v_i and v_j. As before, for any vertex which is not a descendant of v_i or v_j, the probability of the path which uses only the tree edges is the same as it is in the full tree graph. The probability of reaching v and starting at T_i or T_j, using only the tree edges, decreases in the new graph because the subtrees T_i and T_j become smaller. It is now:

$$\sum_{k=1}^{s-i} \frac{(d-2)(d-1)^{k-1}}{nd(d-1)^{k-1}(d-1)^i} + \sum_{k=1}^{s-j} \frac{(d-2)(d-1)^{k-1}}{nd(d-1)^{k-1}(d-1)^j}$$
$$= \frac{(s-i)(d-2)}{nd(d-1)^i} + \frac{(s-j)(d-2)}{nd(d-1)^j}$$

Note that for v_i and v_j the probability of the path using only the tree edges does not change. Now, let us calculate the probability of paths using the edge connecting v_i and v_j. We will now assume that $i = j = 1$, therefore only descendants of v_i and v_j can use this edge, and not any vertex which is a predecessor of v_i or v_j. A vertex at distance k from v_i will need to traverse k edges to reach v_i, take the edge connecting it to v_j and traverse an additional j edges. So we get:

$$\sum_{k=1}^{s-i-1} \frac{(d-2)(d-1)^{k-1}}{nd(d-1)^{k-1}(d-1)(d-1)^i} + \frac{1}{nd(d-1)^i}$$
$$+ \sum_{k=1}^{s-j-1} \frac{(d-2)(d-1)^{k-1}}{nd(d-1)^{k-1}(d-1)(d-1)^j} + \frac{1}{nd(d-1)^j}$$
$$= \frac{(s-i-1)(d-2)}{nd(d-1)^{i+1}} + \frac{1}{nd(d-1)^i} + \frac{(s-j-1)(d-2)}{nd(d-1)^{j+1}} + \frac{1}{nd(d-1)^j}$$

We now calculate the difference in the probability of reaching v: the probability of reaching v in the tree graph minus the probability of reaching v in this new, almost tree graph.

$$\begin{aligned}
\Delta =& \frac{s-i}{nd(d-1)^{i-1}} + \frac{s-j}{nd(d-1)^{j-1}} \\
& - \frac{(s-i)(d-2)}{nd(d-1)^{i}} - \frac{(s-j)(d-2)}{nd(d-1)^{j}} \\
& - \frac{(s-i-1)(d-2)}{nd(d-1)^{i+1}} - \frac{1}{nd(d-1)^{i}} - \frac{(s-j-1)(d-2)}{nd(d-1)^{j+1}} - \frac{1}{nd(d-1)^{j}} \\
=& \frac{s-i}{nd(d-1)^{i}} + \frac{s-j}{nd(d-1)^{j}} \\
& - \frac{(s-i-1)(d-2)}{nd(d-1)^{i+1}} - \frac{1}{nd(d-1)^{i}} - \frac{(s-j-1)(d-2)}{nd(d-1)^{j+1}} - \frac{1}{nd(d-1)^{j}} \\
=& \frac{s-i}{nd(d-1)^{i+1}} + \frac{s-j}{nd(d-1)^{j+1}} \\
& + \frac{d-2}{nd(d-1)^{i+1}} - \frac{1}{nd(d-1)^{i}} + \frac{d-2}{nd(d-1)^{j+1}} - \frac{1}{nd(d-1)^{j}} \\
=& \frac{s-i-1}{nd(d-1)^{i+1}} + \frac{s-j-1}{nd(d-1)^{j+1}}
\end{aligned}$$

Like we have seen in the previous cases, the probability of reaching v decreased after adding an edge connecting v_i and v_j. The amount by which it decreased seems similar to the case in which we added an edge connecting two vertices, such that one is a descendant of the other.

תקציר

בעבודה זו אנו חוקרים את הבעיה של מציאת עותק של תת-גרף H בגרף G, כאשר G רחוק מלהיות חסר עותקים של H. אנו בוחנים את הבעיה הזו במודל של גרפים חסומי דרגה. במודל זה, לכל קודקוד בגרף יש לכל היותר d שכנים והאלגוריתם מורשה לבצע שאילתות לגבי השכנים של הקודקודים בגרף. נאמר כי הגרף G ϵ-רחוק מלהיות חסר עותקים של H אם יש להוריד יותר מ-ϵdn צלעות מ-G על מנת שהוא יהיה חסר עותקים של H.

אנו נציג אלגוריתם למציאת עותק של H בגרפים שהינם ϵ-רחוקים מלהיות חסרי עותקים של H ושה-Treewidth שלהם חסום על ידי קבוע. האלגוריתם מבצע מספר שאילתות פולינומי ב-$1/\epsilon$, בגודל של H ובחסם על הדרגה d. הסיבוכיות של האלגוריתם בלתי תלויה במספר הקודקודים בגרפים, n.

בנוסף, נתאר אלגוריתם למקרה המיוחד בו H הוא מסלול באורך k. האלגוריתם שלנו משתמש בתכונות מסוימות של גרפים שרחוקים מלהיות חסרי מסלולים באורך k. הבעיה הזו נחקרה ע״י אביב רזניק (עבודת מאסטר, מכון ויצמן, 2011). רזניק הציג אלגוריתם למקרה בו הגרף חסר מעגלים והראה כי הסיבוכיות שלו פולינומית ב-$1/\epsilon$, k וב-d. אנחנו מציעים השערה, שאם תוכח, גוררת כי האלגוריתם שלנו עובד על כל גרף שהוא ϵ-רחוק מלהיות חסר מסלולים באורך k עם סיבוכיות פולינומית ב-$1/\epsilon$, k וב-d. הוכחנו את ההשערה הזו עבור גרפים חסרי מעגלים.

אוניברסיטת תל-אביב

הפקולטה להנדסה ע״ש איבי ואלדר פליישמן

בית הספר לתארים מתקדמים ע״ש זנדמן-סליינר

חיפוש תת-גרפים בגרפים חסומי דרגה

חיבור זה הוגש כעבודת מחקר לקראת התואר ״מוסמך אוניברסיטה״ בהנדסת חשמל
על ידי

יניב סבו

העבודה נעשתה בבית הספר להנדסת חשמל
במחלקה למערכות

בהנחיית פרופ׳ דנה רון

חשוון תשע״ז

אוניברסיטת תל-אביב

הפקולטה להנדסה ע״ש איבי ואלדר פליישמן

בית הספר לתארים מתקדמים ע״ש זנדמן-סליינר

חיפוש תת-גרפים בגרפים חסומי דרגה

חיבור זה הוגש כעבודת מחקר לקראת התואר ״מוסמך אוניברסיטה״ בהנדסת חשמל
על ידי

יניב סבו

חשוון תשע״ז