

On the Instance Optimality of Detecting Collisions and Subgraphs

Omri Ben-Eliezer*

Tomer Grossman†

Moni Naor‡

Abstract

Suppose you are given a function $f: [n] \rightarrow [n]$ via (black-box) query access to the function. You are looking to find something local, like a collision (a pair $x \neq y$ s.t. $f(x) = f(y)$). The question is whether knowing the ‘shape’ of the function helps you or not (by shape we mean that some permutation of the function is known). Formally, we investigate the *unlabeled instance optimality* of substructure detection problems in graphs and functions. A problem is $g(n)$ -instance optimal if it admits an algorithm A satisfying that for any possible input, the (randomized) query complexity of A is at most $g(n)$ times larger than the query complexity of any algorithm A' which solves the same problem while holding an *unlabeled copy* of the input (i.e., any A' that “knows the structure of the input”). Our results point to a trichotomy of unlabeled instance optimality among substructure detection problems in graphs and functions:

- A few very simple properties have an $O(1)$ -instance optimal algorithm.
- Most properties of graphs and functions, with examples such as containing a fixed point or a 3-collision in functions, or a triangle in graphs, are $n^{\Omega(1)}$ -far from instance optimality.
- The problems of collision detection in functions and finding a claw in a graph serve as a middle ground between the two regimes. We show that these two properties are $\Omega(\log n)$ -far from instance optimality, and conjecture that this bound is tight. We provide evidence towards this conjecture, by proving that finding a claw in a graph is $O(\log(n))$ -instance optimal among all input graphs for which the query complexity of an algorithm holding an unlabeled certificate is $O\left(\sqrt{\frac{n}{\log n}}\right)$.

1 Introduction

Efficient detection of small structures in complex data is a fundamental challenge across computer science. In this work, we explore to what extent *prior knowledge* on the input may help. Consider, for instance, the problem of detecting a collision in an unknown function $f: [n] \rightarrow [n]$ given query access to f . (Here, a collision in f is a pair of disjoint elements $x \neq y \in [n]$ so that $f(x) = f(y)$.) We ask the following question.

How does an algorithm that knows nothing about f in advance (aside from the domain size n) compare to an algorithm that has some prior knowledge on the structure of f ?

*Simons Institute for the Theory of Computing, University of California, Berkeley, USA. Part of this research was conducted while the author was at Weizmann Institute and later at Massachusetts Institute of Technology. Email: omrib@mit.edu.

†Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: tomer.grossman@weizmann.ac.il.

‡Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: moni.naor@weizmann.ac.il. Supported in part by grant from the Israel Science Foundation (no. 950/16). Incumbent of the Judith Kleeman Professorial Chair.

The prior knowledge we consider in this work takes the form of an *unlabeled copy* of f that the algorithm receives in advance as in Grossman et al. [GKN20]. That is, the algorithm receives a permutation of f – the composed function $f \circ \pi$ for some unknown permutation π – as an “untrusted hint”. We typically call this permutation of f an *unlabeled certificate*; we require the algorithm to be correct with good probability regardless of whether the hint is correct (i.e., even if f is not a permutation of the unlabeled certificate). However, the number of queries made by the algorithm is only measured if the true input is indeed a permutation of the unlabeled certificate.

In the worst case, clearly $\Omega(n)$ queries are necessary, whether we know anything about the structure of f or not. But are there beyond-worst-case instances where holding additional structural information on f may accelerate collision detection?

Definition 1.1 (instance optimality; informal). *A randomized Las Vegas¹ algorithm A deciding if an unknown function $f: [n] \rightarrow [n]$ satisfies a property \mathcal{P} is instance optimal if there exists an absolute constant α satisfying the following. For every function f , and any randomized algorithm A' for the same task, the following holds:*

$$\text{Queries}_A(f) \leq \alpha \cdot \max_{\pi} \text{Queries}_{A'}(f \circ \pi)$$

where the $\text{Queries}_A(\cdot)$ operator refers to the expected number of queries that an algorithm A makes on a certain input.

Finally, we say that \mathcal{P} is instance optimal if there exists an instance optimal algorithm for it.

Note the order of the quantifiers in the definition: for every f , the algorithm A has to compete with an algorithm A' that “specializes” to functions of the form $f \circ \pi$. In other words, an algorithm A is instance optimal if it performs as well as every algorithm A' , that knows the structure of f , but not the actual labels. Note that the correctness of algorithm A' is unconditional – that is A' must be correct even if the structure of f doesn’t match the certificate A' receives.

An algorithm being unlabeled instance optimal means it always performs as well (up to a constant) as the algorithm that knows the structure of the input. If there is no instance optimal algorithm, that means there exists some function where knowing the structure of the function is helpful. Thus, instance optimality is a strong requirement: If a property \mathcal{P} is instance optimal that means that knowing the structure of the input function f *never helps*. When a property is not instance optimal, it will sometimes be useful to discuss its “distance” from instance optimality.

Definition 1.2 (distance from instance optimality; informal). *Consider the setting of Definition 1.1. For a function $\omega(n)$ that grows to infinity as $n \rightarrow \infty$, we say that \mathcal{P} is ω -far from instance optimality if for every algorithm $n \in \mathbb{N}$ and A there exist a function f and an algorithm A' satisfying*

$$\text{Queries}_A(f) \geq \omega(n) \cdot \max_{\pi} \text{Queries}_{A'}(f \circ \pi).$$

Similarly, \mathcal{P} is ω -close to instance optimality if the above inequality holds with \leq instead of \geq .

We may now rephrase our initial question about collisions in the language of instance optimality. Is collision detection an instance optimal problem? I.e., is the property of containing a collision instance optimal? Is it far from instance optimality? Suppose that we have query access to a

¹For simplicity we consider in this paper Las Vegas randomized algorithms, but all of the results apply also to Monte Carlo type algorithms (that allow some error in the returned value).

function $f: [n] \rightarrow [n]$ and are interested in finding a collision. There are two fundamental types of queries to f that one can make: the first option is to query an element x that we have already seen in the past, by which we mean that we have already queried some element y satisfying that $f(y) = x$. This option amounts to extending a “walk” on the (oriented) graph of f . The second option is to query a previously unseen element x , which amounts to starting a new walk. The question, then, is the following: is there a universal algorithm A (which initially knows nothing about f) for choosing when to start new walks, and which walks to extend at any given time, that is competitive with algorithms A' that know the unlabeled structure of f ?

Substructure detection problems. There are many other types of natural problems in computer science that involve small (i.e., constant-sized) substructure detection. A natural generalization of a collision is a k -collision (or multi-collision), where we are interested in finding k different elements x_1, \dots, x_k satisfying $f(x_1) = \dots = f(x_k)$. Fixed points, i.e., values x for which $f(x) = x$, are important in local search and optimization problems, in particular for the study of local maxima or minima in an optimization setting.

Subgraph detection in graphs is also a fundamental problem in the algorithmic literature. Motifs (small subgraphs) in networks play a central role in biology and the social sciences. In particular, detecting and counting motifs efficiently is a fundamental challenge in large complex networks, and a substantial part of the data mining literature is devoted to obtaining efficient algorithms for these tasks. It is thus natural to ask: is it essential to rely on specific properties of these networks in order to achieve efficiency? In other words, are subgraph detection and counting instance optimal problems?

Similarly, the problem of finding collisions is a fundamental one in cryptography. Many cryptographic primitives are built around the assumption that finding a collision for some function, f is hard (e.g. efficiently signing large documents, commitments with little communication and of course distributed ledgers such as blockchain). If one wants to break such a cryptographic system, should one spend resources studying the structure of f ? If finding collisions is instance optimal, that would mean that any attempt to find collisions by studying the structure of a function is destined to be futile.

In this work we focus on the instance optimality of constant-size substructure detection problems in graphs and functions. Before stating our results, let us briefly discuss these data models.

Models. We consider two different types of data access in our work. The first type is that of functions. In this case the input is some function f , and the goal is to determine whether f satisfies a certain property (e.g., whether it contains a collision or a fixed point). In this case the goal of an instance optimal algorithm is to perform as well as an algorithm that receives, as an untrusted hint, the unlabeled structure of the algorithm without the actual assignment of labels. Here the complexity is measured as the number of queries an algorithm makes, where each query takes an input x and returns $f(x)$.

The second type of data is of graphs. Here the goal is to find a constant-sized subgraph. An instance optimal algorithm should perform as well as an algorithm that is given an isomorphism of the graph as an “untrusted hint”. For simplicity, we focus on the standard adjacency list model (e.g., [GRS11]). Here for each vertex the algorithm knows the vertex set V in advance, and can query the identity of the i -th neighbor of a vertex v (for v and i of its choice, and according to some arbitrary ordering of the neighbors), or the degree of v . We note that all of the results also hold in

other popular graph access models, including the adjacency matrix model and the neighborhood query model.

Interestingly, graphs and functions seem closely related in our context. Specifically, the problem of finding a claw in a graph (a star with three edges) is very similar to that of finding a collision in a function, and the results we obtain for these problems are for the most part analogous.

1.1 Main Results and Discussion

Our main result in this paper characterizes which substructure detection problems in functions and graphs are instance optimal. Let us start with the setting of functions.

A structure $H = ([h], E)$ is an oriented graph where each vertex has outdegree at most one, and we say that f contains H as a substructure if there exist values x_1, \dots, x_h such that $f(x_i) = x_j$ if and only if the edge $i \rightarrow j$ exists in H . (For example, a collision corresponds to the structure $([3], \{1 \rightarrow 3, 2 \rightarrow 3\})$.) Finally, the property \mathcal{P}_H includes all functions f containing the structure H . Our first theorem constitutes a partial characterization for instance optimality in functions.

Theorem 1.3 (Instance optimality of substructure detection in functions). *Let H be a connected, constant-sized oriented graph with maximum outdegree 1, and consider the function property \mathcal{P}_H of containing H as a substructure. Then \mathcal{P}_H is*

1. *Instance optimal if $H = P_k$ is a simple oriented path of length k ;*
2. *$n^{\Omega(1)}$ -far from instance optimal for any H that contains a fixed point, two edge-disjoint collisions, or a 3-collision;*
3. *$\Omega(\log n)$ -far from instance optimal for any H that contains a collision.*

Similarly, in graphs we denote by \mathcal{P}_H the property of containing H as a (non-induced) subgraph. Our next theorem provides a characterization for the instance optimality of subgraph detection.

Theorem 1.4 (Instance optimality of subgraph detection in graphs). *Let H be a connected, constant-sized graph with at least one edge. Then \mathcal{P}_H is:*

1. *Instance optimal if H is an edge or a wedge (path of length 2);*
2. *$n^{\Omega(1)}$ -far from instance optimal if H is any graph other than an edge, a wedge, or a claw (a star with 3 edges);*
3. *$\Omega(\log n)$ -far from instance optimal when H is a claw.*

Almost instance optimality of claws and collisions? While we provide a full characterization of those substructures (or subgraphs) H for which \mathcal{P}_H is instance optimal, there remains a notable open problem: is the problem of containing a collision (in functions) or a claw (in graphs) “almost instance optimal”, e.g., is it $O(\log n)$ -close to instance optimality?

The problems of finding a collision in a function and detecting a claw in a graph are closely related and seem to be similar in nature (see Section 3). We conjecture that both of these problems are close to being instance optimal.

Conjecture 1.5. *There exists an algorithm A for collision detection (in functions $f: [n] \rightarrow [n]$) that is $O(\log n)$ -close to instance optimality.*

Conjecture 1.6. *Determining if a graph contains a claw is $O(\log n)$ -close to instance optimality.*

While we are not yet able to prove the conjectures in full generality, we provide initial evidence toward the correctness, at least in the graph case. Specifically, we prove Conjecture 1.6 for graphs in which claw detection is “easy” with a certificate, that is, can be done in up to $O\left(\sqrt{\frac{n}{\log n}}\right)$ queries.

Theorem 1.7 (informal; see Theorem 6.1). *The graph property of containing a claw is $O(\log n)$ -instance optimal when restricted to inputs that require $o\left(\sqrt{\frac{n}{\log n}}\right)$ queries in expectation for an algorithm with an unlabeled certificate.*

While the result was phrased for undirected graphs, it carries on also for collision detection in functions $f: [n] \rightarrow [n]$, in the case where the algorithm is allowed to go both “forward” (i.e., for an x to retrieve $f(x)$) and “backward” (i.e., for x to retrieve elements of the inverse set $f^{-1}(x)$). See the paragraph below on model robustness for further discussion.

We conjecture that the same algorithm we use to show near instance optimality in the regime of Theorem 1.7 is also near instance optimal in the general regime. The algorithm $A_{\text{all-scales}}$ is roughly defined as follows. $A_{\text{all-scales}}$ maintains $m = O(\log n)$ parallel “walks” W_1, \dots, W_m at different “scales”, where in each round (consisting of a total of m queries) $A_{\text{all-scales}}$ adds one step to each of the walks. We try to extend each W_i until it reaches length 2^i or until it has to end (either because of finding a collision/claw or due to reaching the end of a path or closing a cycle). In the case that W_i reaches length 2^i , we “forget” it and restart W_i at a fresh random starting point.

The challenge of merging walks. The only barrier to proving the above two conjectures in the general case seems to be our current inability to deal with “merging” walks in the algorithm. Any algorithm for collision detection in functions, or claw detection in graphs, can be viewed as maintaining a set of walks. In each step we either choose to start a new walk by querying a previously unseen vertex, or extend an existing walk by querying its endpoint (or one of its two endpoints, in the graph case). The event of merging corresponds to the case that two disjoint walks W and W' meet, resulting in the creation of a longer walk $W \cup W'$. Our proof of Theorem 1.7 shows the instance optimality of claw detection in the regime where merging is unlikely to happen during the algorithm’s run.

Model robustness. Throughout the paper we chose to focus on specific models for convenience. However, all our results are model robust and apply in many “natural” models. In particular, in the case of functions we chose to work on the model where an algorithm can only go forward. That is, an algorithm can query $f(x)$ in a black box manner, and doesn’t have the capability to make inverse/backward ($f^{-1}(x)$) queries. Similar characterization results to the graph case also apply if an algorithm can walk backwards; in fact, the model where walking backward is allowed seems to serve as a middle ground between our models for graphs and functions, in the sense that we deal with directed graph properties but are allowed to move in the graph as if it were undirected.

For convenience we wrote all our results for Las Vegas randomized algorithms. All the results in this paper also apply if we require the algorithm to be a Monte Carlo randomized algorithm, i.e., one that is allowed to err with constant probability.

In graphs, we use the popular adjacency list model (which allows sampling random vertices, querying a single neighbor, or querying the degree of a vertex) for data access. The same charac-

terization results also apply under other types of data access, such as the adjacency matrix model or the neighborhood query model (where querying a node retrieves all of its neighbors at once).

1.2 Technical Overview: Collisions and Fixed Points

In this section we give an overview of our main ideas and techniques. Many of the ideas are shared between functions and graphs; we chose to present the main ideas for a few canonical problems, such as fixed point and collision detection in functions, and claw detection in graphs.

Showing the polynomial separation for most graph and function properties amounts, roughly speaking, to providing constructions where a certain substructure is hidden, but where certain hints are planted along the way to help the algorithm holding a certificate to navigate within the graph. Given the constructions, which are themselves interesting and non-trivial, it is not hard to prove the separation. As an example of a polynomial separation construction and result, we discuss the case of a fixed point in functions. For more general statements and proofs regarding these separations, please refer to Sections 4 (for functions) and 5 (for graphs).

The $\Omega(\log n)$ -separation for claws and collisions is the most technically involved “lower bound” type contribution of this paper. Unlike the polynomial separation results, where the core idea revolves around describing the “right” way to hide information, here the construction is more complicated (roughly speaking): the trick of planting hints that allow the algorithm to navigate does not work well, and our arguments rely on the observation that it is sometimes essential for an algorithm without a certificate to keep track of multiple different scales in which relevant phenomena may emerge, compared to an algorithm with a certificate that knows in advance which of the scales is relevant. The proof is also more challenging, requiring us to closely track counts of intermediate substructures of interest. For the sake of the current discussion, we focus on collision detection, but the proof (and construction) for claws is very similar; see Section 3.

Before diving into the ideas behind fixed point and collision detection, let us briefly mention the simplest component in the characterization: an instance optimal algorithm for finding a path of length k . The algorithm chooses a random value and evaluates the function k times on successive values to see if a path of length k emerges (and not a smaller cycle, or a smaller path ending in a fixed point). This is repeated until a path is found or all values have been exhausted. It is instance optimal, since knowing the structure of the function does not help; stopping after less than k steps is meaningless, since it only saves us a constant fraction of the queries.

1.2.1 Fixed point detection: Polynomially far from instance optimality

We give an overview of the proof that finding a fixed point is polynomially far from instance optimality. Small variations of the constructions can be used to show that the same is true for any structure containing a fixed point, a 3-collision, or two edge-disjoint collision.

In order to obtain such a result we provide a distribution of functions that have several fixed points with a secret parameter so that an algorithm with a certificate (knowing the parameter in this case) can find a fixed point in $n^{\frac{3}{4}}$ queries while any algorithm that does not know the secret parameter (i.e. without a certificate) requires $\tilde{\Omega}(n)$ queries to find a fixed point.

The idea is to construct a function f with $\tilde{\theta}(n^{1/4})$ cycles of size roughly $n^{3/4}$, where one random value x in one of the cycles is turned into a fixed point (which effectively turns the said cycle into a path ending at x). It is quite clear that for such a distribution finding the fixed point take time

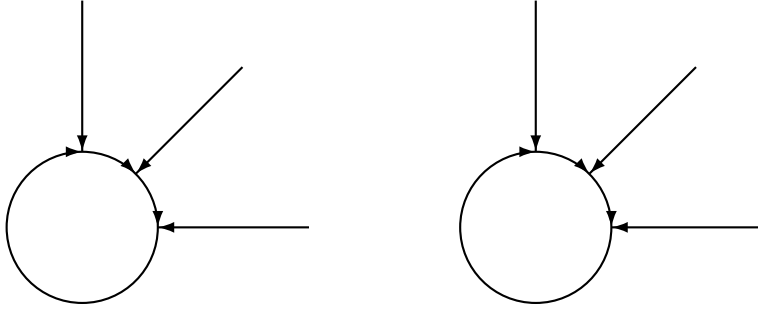


Figure 1: There are $n^{1/4}/\log n$ cycles, where each cycle is of length $n^{3/4}$. Each path entering a cycle is of size $n^{1/4}$. The distance between every two paths on the i -th cycle is p_i .

$\tilde{\Omega}(n)$. But we want to add some information or hint that will allow a certificate holder to find out which is the “correct” cycle.

To give such a hint we add to each cycle many paths of length $n^{1/4}$ entering it. The distance between two paths entering the i th cycle is some (unique) prime p_i where p_i is of size roughly $n^{1/4}$ (so roughly $n^{1/2}$ paths enter the cycle). See Figure 1 for a drawing of this construction.

The hint is the value p_i associated with the unique cycle that ends up with a fixed point. The algorithm (with the hint) we propose will check many (about \sqrt{n}) ‘short’ (length $n^{1/4}$) paths and see when they collide with another set of paths that is supposed to be on the cycles (these are $n^{1/4}$ ‘long’ paths of length \sqrt{n}). Once our algorithm finds three paths entering the same cycle which are of distances that are all a multiple of p_i , the algorithm will conclude that this is the unique path that at its end the fixed point resides and will continue on the path. On the other hand, for any algorithm that does not know which of the p_j ’s is the chosen one and hence the which path ends in a fixed point, each x residing in a cycle is equally likely to be a fixed point, and thus the algorithm requires $\tilde{\Omega}(n)$ queries in expectation.

1.2.2 Finding Collisions: $\Omega(\log n)$ far from instance optimality

The distribution constructed above will not work for collision detection, since functions generated according to this distribution will inherently have many collisions. Below we describe a substantially different construction demonstrating that collision detection is (at least) logarithmically far from instance optimality. We note that the same proof outline, and same construction idea can also be used to show that finding a claw in a graph is not instance optimal.

In order to obtain such a result we provide a distribution of functions that have several collisions, again, with a secret parameter, so that an algorithm with a certificate (knowing the parameter in this case) can find a collision in n^c queries for some constant $c < 1/2$, while any algorithm that does not know the secret parameter (i.e. without a certificate) requires $\Omega(n^c \log n)$ queries to find a collision.

The hard distribution is as follows: there are $\log n$ length scales. For scale i we have $n/2 \cdot 2^i$ cycles, each of length 2^i (note that the total number of nodes in all cycles is $O(n)$). For a uniformly

randomly chosen scale t we turn $n^{1-c}/1.1^t$ of the cycles to be a path ending in a loop of size 2 at the end (this is a collision).

The secret parameter is the value of t . The algorithm with a certificate simply picks a value at random and follows the path it defines for 2^t steps. The algorithm stops if (i) a collision is discovered or (ii) the path has reached length 2^t without a collision or (iii) the path created a cycle of length $2^i < 2^t$. The probability of picking a node on a good path (one ending in a collision of length 2^t) is

$$\frac{n^{1-c} \cdot 2^t}{n \cdot 1.1^t}$$

(since there are $n^{1-c}/1.1^t$ such cycles, each of size 2^t). The cost (in terms of queries) of picking a value on the wrong size cycle, say of size 2^i , is $\min(2^i, 2^t)$. It is possible to show that the total expected work when picking the wrong value is $O(2^t/1.1^t)$.² Therefore the expected amount of work until a good path is found is

$$\frac{2^t}{1.1^t} \cdot \frac{1.1^t \cdot n}{n^{1-c} \cdot 2^t} = n^c.$$

The result is $O(n^c)$ queries in expectation.

We next show that any algorithm that does not know t requires $\Omega(n^c \log n)$ queries, which results in a logarithmic separation from the algorithm with a certificate. In essence, this means that the algorithm needs to spend a substantial effort at all possible scales (instead of just one scale, t) in order to find the collision.

Consider an algorithm without a certificate, and suppose that we choose the secret parameter t in an online manner as follows. Our initial construction starts with $n/2 \cdot 2^i$ cycles of length i . For each such i , we pick $n^{1-c}/1.1^i$ of the cycles of length 2^i , and color one of the nodes in each such cycle by red (call these points the “ i -red points”). Note that at this time we have no information whatsoever on t . Now, each time that a red point on some cycle of length 2^i is encountered, we flip a coin with an appropriate probability (which initially is of order $1/\log n$) to decide whether the current value of i is the secret parameter t or not. If it is, then we turn all i -red points (for this specific value of i) into collisions as described above, and remove the color from all other red points (in paths of all possible lengths). Otherwise, we remove the color from all i -red points (for this specific i) and continue.

It turns out that this construction produces the same distribution as we described before (where t was chosen in advance). However, it can also be shown that to find a collision with constant probability, $\Omega(\log n)$ red points need to be encountered along the way. The rest of the analysis provides an amortized argument showing that the expected time to find each red vertex by any algorithm is $\Omega(n^c)$. The main idea of the amortized analysis (which we will not describe in depth here) is to treat cycles in which we made many queries – at least a constant fraction of the cycle – differently from cycles where we made few queries. For cycles of the first type, the probability to find a red point (if one exists) is of order $2^i/n^c$, but the amount of queries that we need to spend is proportional to 2^i . For cycles of the second type, each additional query only has probability $O(1/n^c)$ to succeed finding a red point, but the query cost is only 1. In both cases, the rate between the success probability and the query cost is of order $1/n^c$.

²the constant 1.1 is a bit arbitrary, and other constants larger than 1 will also work.

1.2.3 Finding claws: $O(\log n)$ -close to instance optimality in merging-free regime

The proof that collision detection is $\Omega(\log n)$ -far from instance optimality extends very similarly to claw detection in graphs. We next show that this bound is tight in the “low query” regime where G admits an algorithm for claw detection (with a certificate) using $q \leq \alpha \sqrt{\frac{n}{\log n}}$ queries, for a small constant $\alpha > 0$.

Every claw-free graph is a union of disjoint cycles and paths. Thus, every algorithm for finding a claw can be viewed as maintaining a set of walks of several types: Some of the walks may have closed a cycle; others may have reached one or two ends of the path P that the walk resides in. All other walks simply consist of consecutive vertices in the interior of some path in G .

Clearly, walks of the first type – those that have closed a simple cycle – cannot be extended to find a claw. Our first part of the proof is a reduction eliminating the need to consider walks of the second type (i.e., ones that have reached at least one endpoint). Specifically, we show that for every graph G on n vertices there is a graph G' on $n + 2$ vertices satisfying several properties:

- In all simple paths in G' , either both endpoints have degree 1 or both are degree 3 or more (i.e., are claw centers).
- The query complexity of detecting a claw in G' is equal, up to a multiplicative constant, to that of G . This is true both with or without an unlabeled certificate.
- The construction of G' from G is deterministic. Thus, an unlabeled certificate for G' can be constructed if one has an unlabeled certificate for G .

The construction is very simple: we add two new vertices and connect them to all degree-1 vertices (and to each other, if needed).

Merging without claws requires $\Omega\left(\sqrt{\frac{n}{\log n}}\right)$ queries. The second part of our argument shows that one cannot make two walks merge in the first $\alpha \sqrt{n/\log n}$ queries (with constant probability and for some small constant $\alpha > 0$) without finding a claw beforehand. The proof relies on an advanced birthday paradox type analysis that is suitable for adaptive algorithms. For the purpose of this part, one may consider G as a union of paths and cycles (without any claws). We bucket these paths and cycles into $O(\log n)$ groups, where in each group all paths and cycles are of roughly the same length – within a multiplicative factor of 1.1 from each other. Suppose that after each “new” (previously unseen) vertex is queried, the algorithm immediately knows to which bucket this vertex (and the corresponding walk emanating from it) belongs. We show that even in this case the lower bound holds.

Focusing on a specific bucket, let \mathcal{W} be the set of all walks in this bucket at some point in the algorithm’s run. An assignment of walks to “locations” within the bucket is considered valid if no two walks intersect. We argue that the set of locations of walks is uniformly random among all sets of valid configurations. Similarly to our analysis of the $\Omega(\log n)$ separation (see the previous subsection), our next step in this part deals separately with “short walks” and “long walks”. Very roughly speaking, our proof shows that walks have sufficient “degree of freedom” so that their probability to merge will be very small, even if provided that they lie in the same bucket. We omit the precise details of the analysis from this overview, and point the reader to Section 6.2.

Asymptotic stochastic dominance of $A_{\text{all-scales}}$ The third and last part of the argument shows that the algorithm $A_{\text{all-scales}}$ mentioned in Section 1.1 stochastically dominates any other algorithm (asymptotically) in the following sense. Conditioned on an algorithm A (making $q = O\left(\sqrt{\frac{n}{\log n}}\right)$ queries) not encountering any merging walks during its operation, the algorithm $A_{\text{all-scales}}$ is at least as likely to find a claw, while using a slightly larger amount of roughly $4q \log n$ queries.

Recall first how $A_{\text{all-scales}}$ is defined. For $0 \leq i < \log n$ let $A^{(i)}$ be the algorithm which repeatedly does the following: pick a random vertex in G ; make a bidirectional walk from it for 2^{i+1} steps, or until a claw is found (leading to a “win”) or an endpoint is found (leading to an early termination). $A_{\text{all-scales}}$ maintains one copy of each $A^{(i)}$ (for a total of $\log n$ copies), and alternates between them: each round of $A_{\text{all-scales}}$ makes $\log n$ queries, one for each $A^{(i)}$.

Now let A be any algorithm operating on graphs on n vertices. Consider the event $E_i = E_i(G, q, A)$ that A finds a claw (for the first time) after at most q queries through the following process. A queries a “new” vertex v of distance between 2^i and $2^{i+1} - 1$ from the claw center w , and finds it by completing a walk from v to w . We claim that the event that A finds a claw is equal to the union of the events E_i (for $0 \leq i \leq \log n$).

The proof goes through a careful coupling argument between A and any fixed $A^{(i)}$ (separately). Through the coupling, we may assume that A and $A^{(i)}$ have access to the same source of randomness generating “new” vertex queries, that is, the j -th new vertex starting a walk W_j in A is also the j -th new vertex in $A^{(i)}$. We may further assume, by symmetry considerations, that A respects an “older first” principle: if two walks W and W' have exactly the same “shape” (within G) at some point, then A will prefer to extend the walk among them that is older. Now, suppose that A finds the claw in its walk W_j , of distance between 2^i and $2^{i+1} - 1$ from v_j . By the “older first” principle, this implies that for all vertices $v_{j'}$ with $j' < j$ that are at least 2^i away from an endpoint (call these values of j' good), A must have walked for at least 2^i from $v_{j'}$ so far. For all other values of $j' < j$ (call these bad), A walked a number of steps that is at least the distance from an endpoint. In contrast, $A^{(i)}$ makes up to $4 \cdot 2^i$ queries around each good vertex, and up two times as many queries as A around each bad one.

1.3 Related Work

The term instance optimality was coined by Fagin, Lotem and Naor [FLN03]. If an algorithm always outperforms every other algorithm (up to a constant), particularly the algorithm that is aware of the input distribution, it is defined as being instance optimal. This definition is very strict, and thus there are numerous situations in which it cannot be meaningfully attained. As a result, several works (including ours) address this by necessitating that an instance optimal algorithm be competitive against a certain class of algorithms (for example, those that know an a permutation of the input, rather than the input itself). This notion of instance optimality is sometimes referred to as “unlabeled instance optimality”.

Unlabeled instance optimality. Afshani, Barbay, and Chan [ABC17] considered and presented unlabeled instance-optimal algorithms for locating the convex hull and set maxima of a set of points. Valiant and Valiant [VV16] developed an unlabeled instance optimal algorithm for approximating a distribution using independent samples from it (with the cost function being the number of samples). Later [VV17], they provided an algorithm for the identity testing problem. Here the problem is determining, given an explicit description of a distribution, whether a collection of

samples was selected from that distribution or from one that is promised to be far away. More recent works on instance optimality in distribution testing include, for example, the work of Hao et al. [HOSW18, HO20].

Grossman, Komargodski, and Naor examined unlabeled instance optimality in the query model [GKN20]. Their work relaxes the definition of instance optimality by no longer requiring an optimal algorithm to compete against an algorithm that knows the entire input, but rather against an algorithm that knows *something* about the input. Arnon and Grossman [AG21] define the notion of min-entropic optimality, where instead of relaxing the “for-all” quantifier over algorithms, they relax “for-all” quantifier over inputs. That is, for an algorithm to be optimal it is still required to perform as well as any other algorithm; however it is no longer required to be optimal on every input distribution, but rather only on a certain class of inputs.

Instance optimality in graphs. Subgraph detection and counting has not been thoroughly investigated from the perspective of instance optimality; establishing a unified theory of instance optimality in this context remains an intriguing open problem. However, instance optimality has been investigated for other graph problems of interest. For example, Haeupler, Wajc and Zuzic [HWZ21] investigate instance optimality and a related notion called universal optimality in a family of classical and more “global” distributed graph problems such as computing minimum spanning trees and approximate shortest paths.

Strong instance optimality. The original, robust definition of instance optimization calls for an algorithm to be superior to every other algorithm. For getting the top k aggregate score in a database with the guarantee that each column is sorted, [FLN03] provided an instance-optimal algorithm. Demaine, López-Ortiz, and Munro [DLM00] provided instance-optimal algorithms for locating intersections, unions, or differences of a group of sorted sets. Baran and Demaine [BD04] showed an instance optimal algorithm for finding the closest and farthest points on a curve. Grossman, Komargodski and Naor [GKN20] and Hsiang and Liu [LM23] studied instance optimality in the decision tree model.

Cryptography and complexity. The problems of finding a constant sized structure in f , where f is a total function guaranteed to contain the structure at hand has been studied extensively and is a fundamental problem in computational complexity and there are complexity classes in TFNP defined around it [MP91, Pap94]. We note that we can slightly change the functions in our paper to also make them total problems, and all our proofs will still hold.

As mentioned above, the problem of finding collisions is a fundamental one in cryptography. The standard definition is that of a collision resistant hash (CRH), where finding a collision is a computationally hard problem. Such functions are very significant for obtaining efficient signature schemes and commitment to large piece of data using little communication. But other related structures are also considered in the literature: for instance, functions where it is hard to find multiple collisions [KNY18].

1.4 Organization

In Section 2 we formally define the computational models we use as well as the notions of an unlabeled certificate (i.e., the “untrusted hint”) and instance optimality. In Section 3 we prove that

finding a collision in functions, and a claw in a graph is not instance optimal. In Section 4 we prove that functions that are not subsets of the collisions of two paths, followed by an additional path is polynomial far from being instance optimal. Such properties include many fundamental structures such as finding a fixed point, or a multi-collision. This gives us a complete characterization in the function model. In Section 5 we complete the full characterization for the model defined on graphs. We do this by proving that finding a path of length one or two is instance optimal, and that determining if a graph contains a subgraph H is polynomially far from being instance optimal, unless H is a path of length 1 or 2, or a claw. Finally, in Section 6 we prove that finding a claw in a graph is $O(\log(n))$ close to being instance optimal among graphs for which finding a claw or collision can be done in $O(\sqrt{n}/\log(n))$, by an algorithm with an unlabeled certificate.

2 Preliminaries

2.1 Functions

Let $n \in \mathbb{N}$. A *property* \mathcal{P} of functions is a collection of functions $f: [n] \rightarrow [n]$ that is closed under relabeling. That is, if $f \in \mathcal{P}$ then $f \circ \pi \in \mathcal{P}$ for any permutation $\pi: [n] \rightarrow [n]$. We sometimes say that f *satisfies* \mathcal{P} when $f \in \mathcal{P}$.

In this work we will be interested in the property of containing some constant size substructure H (e.g., a collision or a fixed point). Let H be an oriented graph with h vertices. Suppose further that the outdegree of each vertex in H is at most 1.³ The property \mathcal{P}_H consists of all functions $f: [n] \rightarrow [n]$ satisfying the following. There exist h disjoint elements $x_1, \dots, x_h \in [n]$ and a mapping between $V(H)$ and $\{x_1, \dots, x_h\}$, so that H contains an edge between u and v if and only if $f(x_u) = x_v$, where x_u, x_v are the mappings of u and v , respectively.

A Las Vegas (randomized) algorithm for the property \mathcal{P} in the query model is a randomized decision tree that determines membership in the property \mathcal{P} with probability 1 (i.e., it is always correct, and the quantity of interest is the number of queries the algorithm requires). Given a Las Vegas randomized algorithm A (which knows n) with random seed r and given $f: [n] \rightarrow [n]$, denote by $\text{Queries}_A^{\mathcal{P}}(f, r)$ the amount of queries that A makes when evaluating if f satisfies a property using the random seed r . Usually when the property \mathcal{P} is clear from context we omit it from the notation.

We write $\text{Queries}_A^{\mathcal{P}}(f) = \mathbb{E}_r \text{Queries}_A^{\mathcal{P}}(f, r)$ (or $\text{Queries}_A(f)$) to denote the expected number of queries that the algorithm A makes over input f , where the expectation is taken over all possible random seeds.

Definition 2.1 (Unlabeled Certificate Complexity). *The Unlabeled Certificate complexity of a property \mathcal{P} , and function f is:*

$$\text{RAC}(\mathcal{P}, f) = \min_{A \in \mathcal{A}_{\mathcal{P}}} \max_{\pi} \text{Queries}_A(f \circ \pi),$$

where $\mathcal{A}_{\mathcal{P}}$ is the set of all Las Vegas algorithms for evaluating if f satisfies property \mathcal{P} , and π ranges over all permutations of $[n]$.

So far, we have considered only properties of functions of a given size n . Our definition of instance optimality is asymptotic in its nature and so we extend the definition of a property by

³Note that a function can be viewed as an oriented graph where the outdegree is always equal to one, hence H can appear as a substructure in such a function if and only if the outdegrees are at most 1.

allowing it to have functions of different sizes. Suppose that \mathcal{P} is a property which contains graphs of all sizes $n \geq N$, for some constant N . We can then define a corresponding sequence of algorithms $\{A_n\}_{n \geq N}$, where A_n is responsible for graphs of size n .

Definition 2.2 (instance optimality). *A sequence of properties $\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$ invariant under a relabeling is instance optimal if there exist an absolute constants $c > 0$, and a sequence $\mathcal{A} = \{A_n\}_{n \in \mathbb{N}}$, where each A_n is a Las Vegas algorithm for \mathcal{P}_n , such that on every input $f: [n] \rightarrow [n]$, it holds that*

$$\text{Queries}_{A_n}^{\mathcal{P}_n}(f) \leq c \cdot \text{RAC}(\mathcal{P}_n, f)$$

Next we present the analogous definition for being far from instance optimality.

Definition 2.3 (ω -far from instance optimality). *Let $\omega: \mathbb{N} \rightarrow \mathbb{N}$ denote a function that grows to infinity as $n \rightarrow \infty$. We say that a sequence of algorithms $\{A_n\}_{n \in \mathbb{N}}$ evaluating if a sequence of functions $\{f_n\}_{n \in \mathbb{N}}$ (where $f_n: [n] \rightarrow [n]$) satisfies a sequence of properties $\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$ is ω -far from instance optimal if there exists a constant N where for all $n \geq N$ it holds that:*

$$\text{Queries}_{A_n}^{\mathcal{P}_n}(f_n) \geq \omega(n) \cdot \text{RAC}(\mathcal{P}_n, f_n).$$

We say that the sequence of properties $\{\mathcal{P}_n\}$ is ω -far from instance optimal if any sequence of algorithms $\{A_n\}_{n \in \mathbb{N}}$ evaluating it is ω -far from instance optimal.

In particular, a property \mathcal{P} (or more precisely a sequence $\{\mathcal{P}_n\}$ of properties of functions $f: [n] \rightarrow [n]$, for any n) is polynomially far from instance optimal, if it is ω -far for some $\omega(n) = n^{\Omega(1)}$ polynomial in n .

2.2 Graphs

A graph property \mathcal{P} is a collection of graphs that is closed under isomorphism. That is, if $G = (V, E) \in \mathcal{P}$ and $\pi: V \rightarrow V$ is a permutation, then the graph $G^\pi = (V, E^\pi)$ where $(u, v) \in E$ if and only if $(\pi(u), \pi(v)) \in E^\pi$ satisfies $G^\pi \in \mathcal{P}$.

Here we consider the adjacency list query model. We assume that the vertex set V is given to us in advance. Given a single query, an algorithm can either (i) find the degree d_v of v , or (ii) find the i -th neighbor of v (in some arbitrary ordering). We note that other variants of the adjacency list model in the literature also allow pair queries, that is, given $u, v \in V$ the algorithm can ask whether there is an edge between u and v . Our results hold word for word also in this variant.

Definitions of instance optimality are analogous to Section 2.1, except here the unlabeled certificate is an isomorphism of the graph.

Definition 2.4 (Unlabeled certificate complexity). *The **randomized unlabeled certificate complexity** of a graph property \mathcal{P} with respect to a graph G is defined as follows.*

$$\text{RAC}(\mathcal{P}, G) = \min_{A \in \mathcal{A}_{\mathcal{P}}} \max_{\pi \in \Gamma} \text{Queries}_A^{\mathcal{P}}(\pi(G)),$$

where Γ is the set of all permutations of the vertex set, and $\mathcal{A}_{\mathcal{P}}$ is the set of all Las Vegas randomized algorithms that always evaluate membership in \mathcal{P} correctly.

Definition 2.5 (instance optimality). *A sequence of graph properties $\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$ is instance optimal if there exist a constant $c > 0$ and a sequence of Las Vegas randomized algorithms $\mathcal{A} = \{\mathcal{A}_n\}_{n \in \mathbb{N}}$ for \mathcal{P} , such that on every input G on n vertices, it holds that*

$$\text{Queries}_{\mathcal{A}_n}^{\mathcal{P}_n}(G) \leq c \cdot \text{RAC}(\mathcal{P}_n, G)$$

Definition 2.6 (ω -far from instance optimality). *Let $\omega: \mathbb{N} \rightarrow \mathbb{N}$ denote a function that grows to infinity as $n \rightarrow \infty$. A sequence of algorithms $\{\mathcal{A}_n\}_{n \in \mathbb{N}}$ evaluating if a sequence of graphs $\{G_n\}_{n \in \mathbb{N}}$ (where G_n is a graph of order n) satisfies a sequence of properties $\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$ is ω -far from instance optimal if there exists a constant N where for all $n \geq N$ it holds that:*

$$\text{Queries}_{\mathcal{A}_n}^{\mathcal{P}_n}(G_n) \geq \omega(n) \cdot \text{RAC}(\mathcal{P}_n, G_n).$$

We say that the sequence of properties $\{\mathcal{P}_n\}$ is ω -far from instance optimal if any sequence of algorithms $\{\mathcal{A}_n\}_{n \in \mathbb{N}}$ evaluating it is ω -far from instance optimal.

We conclude with standard graph theory terminology. A simple path with $k \geq 2$ vertices (in an undirected graph) is a collection of disjoint vertices v_1, \dots, v_k where v_i is connected to v_{i+1} by an edge for each $i = 1, 2, \dots, k-1$. A simple cycle is defined similarly, but with v_k also connected to v_1 . Finally, the claw graph plays a central role in this work.

Definition 2.7 (Claw). *The Claw graph, S_3 , is a 3-star. That is, a four vertex graph consisting of a single vertex, of degree three, which is connected to three vertices each with degree one.*

3 Collisions and Claws: Logarithmic Separation

In this section we formally present and analyze our construction proving Theorem 1.3 Item 3 and Theorem 1.4 Item 3: that detecting collisions (in functions $f: [n] \rightarrow [n]$) and claws (in graphs) is not instance optimal.

Theorem 3.1 (Theorem 1.4 Item 3 Reworded). *The property \mathcal{P}_{S_3} of containing a claw is $\Omega(\log(n))$ -far from instance optimality.*

Theorem 3.2 (Theorem 1.3 Item 3 Reworded). *Fix $a, b, c \in \mathbb{N}$. Let $H = H_{a,b,c}$ denote the oriented graph containing two paths of length a and b which collide in a vertex, followed by a path of length c . The function property \mathcal{P}_H is $\Omega(\log(n))$ -far from instance optimal.*

These two cases (i.e., claws in graphs and collisions in functions) are very similar and the proof that they are not instance optimal is almost identical. Thus, for the majority of the section we focus on the case of claws in graphs. At the end of the section we describe the minor adaptations required for the case of collisions in functions.

We start by presenting the construction for claws. In Section 3.1 we adapt the construction for collisions in functions. Here and in the rest of the paper, we do not try to optimize the constant terms. In particular, the constant $c = 1/10$ appearing in the exponent of the query complexity is somewhat arbitrary; the same construction essentially works for any $c < 1/2$ (and with some adaptations it can be made to work for larger values of the constant c).

Construction 3.3. *Consider the following process for generating a graph over the vertex set $[n]$, which starts with an empty graph and gradually adds edges to it.*

- For each integer $\frac{1}{1000} \log n \leq i \leq \frac{1}{100} \log n$, pick $a_i = n/2.2^i$ uniformly random disjoint simple paths of length 2^i in the graph.
- Pick a uniformly random integer $\frac{1}{1000} \log n \leq t \leq \frac{1}{100} \log n$, which we consider as the “good” index. Pick a random collection \mathcal{P}_t of $b_t = n^{9/10}/1.1^t$ of the paths of length 2^t . Apply to each path $P \in \mathcal{P}_t$ the following transformation: let u_P and v_P denote the two ends of the path. Now connect u_P to two isolated vertices, and v_P to two other isolated vertices. This turns P into a tree of size $2^t + 4$ built from a long path and two claws, one at each end of the path.
- All vertices that do not participate in any of the above structures remain isolated.

We claim that an algorithm holding a certificate requires only $O(n^{1/10})$ queries to find a claw. Since t is known from the certificate, the strategy is simply to only try walks of length 2^t .

Lemma 3.4. $\mathbb{E}_{G \leftarrow \Delta} \text{RAC}(\mathcal{P}, G) = O(n^{1/10})$.

Proof. **TOPROVE 0** □

The main result of this section, given below, is a lower bound showing that algorithms without a certificate require a number of queries that is larger by a multiplicative logarithmic factor compared to the best algorithm with a certificate.

Theorem 3.5. For any algorithm \mathcal{A} (without a certificate), $\mathbb{E}_{G \leftarrow \Delta} \text{Queries}_{\mathcal{A}}(G) = \Omega(n^{1/10} \log(n))$.

To prove Theorem 3.5, we first revisit Construction 3.3, discussing an equivalent way to generate the same distribution that is more suitable for our analysis. This alternative construction has some offline components, that take place before the algorithm starts to run, and an online component, that reveals some of the randomness during the operation of the algorithm.

For what follows, let $B(n)$ denote the maximum possible number of (initially isolated) vertices that are added as neighbors of claws in the second part of Construction 3.3. Note that this number is maximized when t takes its minimal possible value, and satisfies $B(n) \leq n^{9/10}/1.1^{\log(n)/1000} = n^{9/10 - \Omega(1)}$.

Unlike the original construction, here we think of the vertices of the constructed graph as having one of three colors: black, red, or blue. These colors shall help us keep track of the analysis. In essence (and roughly speaking), blue vertices lead to an immediate victory but they are very rare and unlikely to be found in less than $n^{1/10 + \Omega(1)}$ queries; red vertices are not as rare: finding one of these takes roughly $n^{1/10}$ queries, but $\Omega(\log n)$ red vertices are required to find a claw with constant probability; finally, all other vertices are black, and encountering a black vertex contributes very little to the probability of finding a claw.

Construction 3.6. We start with an empty graph on n vertices colored black, and color $B(n)$ of the vertices in blue.

We then construct, as in the first bullet of Construction 3.3, $n/2.2^i$ disjoint paths of length 2^i out of black vertices only, for every $\frac{1}{1000} \log n \leq i \leq \frac{1}{100} \log n$.

Next, for every i we pick a subset of $b_i = n^{9/10}/1.1^i$ paths out of those of length 2^i . We color the ends of these paths in red.

The last part of the construction happens online, while the algorithm runs. In each time step where the algorithm visits a black vertex, the construction remains unchanged. If the algorithm encounters a red vertex, then we reveal the randomness in the construction in the following way.

- Let $I = \{\frac{1}{1000} \log n \leq i \leq \frac{1}{100} \log n : \text{there exists a path of length } 2^i \text{ with a red end}\}$. Note that initially, I simply contains all values of i in the relevant range; however in the construction I will become smaller with time.
- Let $i \in I$ denote the unique integer satisfying that the currently visited red vertex lies on a path of length 2^i . We flip a coin with probability $1/|I|$. If the result is ‘heads’, we consider i as the “good” index and do the following: all red ends of paths of length 2^i are connected to (isolated) blue vertices, all vertices in the graph are recolored by black, and the construction of the graph is complete.
- If the result of the above flip is ‘tails’, we turn all red ends of paths of length 2^i to black, and remove i from I .

Finally, if the algorithm encounters a blue vertex, we pick $i \in I$ uniformly at random to be the “good” length, and connect the red ends of paths of length 2^i to blue isolated vertices randomly. We then recolor all vertices in the graph to black and consider the construction complete.

It is straightforward to check that Construction 3.6 produces the exact same distribution over graphs as Construction 3.3, and furthermore it does not reuse randomness revealed by the algorithm in previous parts. Of particular interest is the following observation.

Observation 3.7. *Consider any point of time during the online phase of Construction 3.6, and let I be as the defined in the first bullet. Then for any $t \in I$, the probability that t will be the eventual “good” index, conditioning on all previous choices made during the construction, is $1/|I|$.*

We say that \mathcal{A} wins if it either finds a claw or encounters a blue vertex. The following lemma is a strengthening of Theorem 3.5, and its proof immediately yields a proof for the theorem.

Lemma 3.8. *There exists $C > 0$ such that for any $n \in \mathbb{N}$, any algorithm without a certificate requires at least $Cn^{0.1} \log n$ queries to win with success probability $9/10$.*

From this lemma, it immediately follows that the expected winning time for the algorithm is $\Omega(n^{0.1} \log n)$, which in turn implies the theorem (by definition of winning). Indeed, any algorithm that finds a claw can immediately find a blue vertex and win, since each claw center has two blue neighbors. Thus, we devote the rest of this section to the proof of Lemma 3.8.

The next (easy) lemma states that encountering a blue vertex is a rare event which will not substantially impact our analysis.

Lemma 3.9. *There exists an absolute constant $\varepsilon > 0$ satisfying the following. For any algorithm \mathcal{A} without a certificate, with high probability \mathcal{A} does not query a blue vertex within $n^{\frac{1}{10} + \varepsilon}$ steps.*

Proof. **TOPROVE 1** □

The following result shows that finding $\Omega(\log n)$ red vertices with constant probability requires $\Omega(n^{0.1} \log n)$ queries. To complete the proof, we shall see later that either finding a blue vertex or collecting at least logarithmically many red vertices is essential to win with constant probability.

Lemma 3.10. *There exists a constant $c > 0$ so that for all $n \in \mathbb{N}$, any algorithm \mathcal{A} which makes $cn^{0.1} \log n$ queries will find less than $\frac{1}{1000} \log n$ red vertices in expectation.*

Proof. **TOPROVE 2** □

We now have all the ingredients to complete the proof of Lemma 3.8.

Proof. **TOPROVE 3** □

3.1 From Claws to Collisions

We now briefly define a similar construction aimed at showing the $\Theta(\log n)$ separation for collisions in functions. The same construction and proof also apply if instead of a collision we consider an “extended collision” $H_{a,b,c}$ as defined in Theorem 3.2.

Construction 3.11. *Consider the following process for generating a function $f: [n] \rightarrow [n]$:*

- *For each integer $\frac{1}{1000} \log n \leq i \leq \frac{1}{100} \log n$, add $a_i = n/2.2^i$ disjoint paths of length 2^i in the function.*
- *Pick a uniformly random integer $\frac{1}{1000} \log n \leq t \leq \frac{1}{100} \log n$. Pick a collection \mathcal{P}_t of $b_t = n^{9/10}/1.1^t$ of the paths of length 2^t . For each such path P , let u and v denote the first and last element in the path; set $f(v)$ to be an arbitrary value in $P \setminus \{u, v\}$, which creates a collision. Close all other paths (of all lengths) into cycles: specifically, using the same notation, set $f(v) = u$.*
- *For all values $x \in [n]$ that do not participate in any of the above paths, set $f(x) = x$ to be a fixed point.⁴*

As is the case with claws in graphs, an algorithm holding a certificate would know the value of t , and make walks of length 2^t until finding a collision, with a query complexity of $O(n^{1/10})$. Meanwhile, to show the lower bound for an algorithm without a certificate, we use a coloring scheme with only two colors – red and black – where elements that are ends of paths which serve as “candidates” for a collision are marked red, and all other elements are marked black. Similarly to the above, in order to find a collision with constant probability, the algorithm needs to find $\Omega(\log n)$ red elements with constant probability, which requires $\Omega(n^{1/10} \log n)$ queries.

4 Function Properties Far from Instance Optimal

In this section we study the instance optimality of properties of functions $f: [n] \rightarrow [n]$. As shown in the previous section, the property of containing a collision is $\Theta(\log n)$ -far from instance optimal. We show that any pattern with either at least two collisions or at least one fixed point is *polynomially* far from instance optimality. This is summarized in the theorem below.

Theorem 4.1. *Let H be any constant-size oriented graph (possibly with self-edges) where each node has out-degree at most one. Suppose further that H either contains (i) a fixed point (i.e., an edge from a node to itself) or (ii) at least two nodes with in-degree at least two, or (iii) at least one node with in-degree at least three.*

The function property \mathcal{P}_H of containing H as a substructure is $\tilde{\Omega}(n^{1/4})$ -far from being instance optimal.

The rest of this section is devoted to the proof of the theorem. We start with the construction used to prove the theorem.

⁴We note that the construction can be easily modified to not include fixed points: simply use the remaining values to close cycles of length 2 or 3 instead of fixed points, which are essentially cycles of length 1.

Construction 4.2. Let H be an oriented graph satisfying the conditions of Theorem 4.1. Define the entry vertices of H to be those vertices with in-degree 0 in H (in the special case where H is a single fixed point, define its single vertex as the entry point). Let T be the total number of entry vertices in H .

Define an input distribution Δ as follows: first, pick $\alpha \frac{n}{\log n}$ vertices uniformly at random and split them into $N = \alpha n^{1/4} / \log(n)$ disjoint cycles of length $n^{3/4}$, for a small absolute constant $\alpha > 0$. Denote these cycles by C_1, \dots, C_N . For each cycle C_i we associate a unique prime number, p_i , where all p_i 's are in the range $(n^{1/4}/4, n^{1/4}/2)$ for an appropriate value of c . Note that this is possible due to well-known results on the density of prime numbers. We say that all points contained in the union $\bigcup_{i=1}^N C_i$ are of type 1.

For each cycle C_i , we add paths of length $\alpha n^{1/4}$ entering it, where the distance (in C_i) between the entry points of every two adjacent paths entering the cycles is exactly p_i . Since each cycle C_i has a length of $n^{3/4}$, it has $\Theta(\sqrt{n})$ paths entering it, each of length $n^{1/4}$. We say that all points participating in these paths are of type 2.

Lastly, a collection x_1, \dots, x_T of exactly T points from $\bigcup_{i=1}^N C_i$ is picked uniformly at random conditioned on the event that no two of these points come from the same cycle. Denote the latter event by E and note that $\Pr(E) = 1 - o(1)$. For each x_k , let C_{i_k} denote the cycle containing it, and turn this cycle into a path ending at x_k by removing the outgoing edge from x_k . Finally, insert a copy of H using all x_1, \dots, x_T as entry points.

To complete the function into one that has size n , partition all remaining (unused) points into disjoint cycles of length $n^{3/4}$ each.

Crucially, an algorithm with a certificate knows the indices i_1, \dots, i_T (and the corresponding primes p_{i_1}, \dots, p_{i_T}). Given these primes, it turns out that the algorithm is able to find the relevant cycles, walk on them until finding the entry points, and building the full H -copy, all using $O(n^{3/4})$ queries. In contrast, for an algorithm without the certificate, the entry points are distributed uniformly over the union of all cycles, and thus a lower bound of $\tilde{\Omega}(n)$ can be shown.

Lemma 4.3. $\mathbb{E}_{f \leftarrow \Delta} \text{RAC}(\mathcal{P}_H, f) = O(n^{3/4})$.

Lemma 4.4. For any algorithm \mathcal{A} (without a certificate), $\mathbb{E}_{f \leftarrow \Delta} \text{Queries}_{\mathcal{A}}(f) = \Omega(n / \log n)$.

Proof. [TOPROVE 4](#) □

Proof. [TOPROVE 5](#) □

5 Instance Optimality in Graphs

We consider whether questions of the type: “Does G contains a subgraph H ” are instance optimal. We begin by defining a k -star.

Definition 5.1 (k -star). The k -star S_k is a graph with $k + 1$ vertices: k vertices of degree 1, all connected to a vertex of degree k .

For example, a 1-star is a graph consisting of 2 vertices, connected by an edge. A 2-star (or a wedge) contains 2 vertices of degree 1 connected to a third vertex of degree 2. A 3-star is a claw.

In Section 3, we have seen that claws are not instance optimal, by showing a $\Omega(\log n)$ -separation between an algorithm with a certificate and one without a certificate in some case. What about

other choices of H ? It turns out that if H is a 1-star or 2-star then finding H is instance optimal. If H is any other graph, then finding H is polynomially far from instance optimal.

Theorem 5.2 (Theorem 1.4 repeated). *Let H be any fixed graph and consider the property \mathcal{P}_H of containing a copy of H as a subgraph. Then \mathcal{P}_H is:*

- *instance optimal if H is an edge or a wedge (path with two edges);*
- *$n^{\Omega(1)}$ -far from instance optimal if H is any graph other than an edge, a wedge, or a claw; and*
- *$\Omega(\log(n))$ -far from instance optimal when H is a claw.*

The theorem follows from the lemmas below, together with the results of Section 3. Lemma 5.3 proves the first item of the theorem. Lemma 5.4 proves that for every H that is not a k -star, finding H is not instance optimal. Lemma 5.5 proves the separation for k -stars when $k \geq 4$.

Lemma 5.3. *\mathcal{P}_H is instance optimal when H is a 1-star (edge) or a 2-star (wedge).*

Lemma 5.4. *For all graphs H that are not a k -star, there exists a distribution, Δ , such that $\mathbb{E}_{G \leftarrow \Delta} \text{Queries}_A(G) = \Omega(n)$ for any algorithm A (without a certificate) determining membership in \mathcal{P}_H , whereas $\text{RAC}(\mathcal{P}_H, \Delta) = O(n^{1/2} \log n)$.*

Lemma 5.5. *Let $H = S_k$ for $k \geq 4$. There exists a distribution Δ such that for any algorithm A (without a certificate) determining membership in \mathcal{P}_H , $\mathbb{E}_{G \leftarrow \Delta} \text{Queries}_A(G) = \Omega(n)$, while $\text{RAC}(\mathcal{P}_H, \Delta) = O(n^{1/2})$.*

Proof. **TOPROVE 6** □

Proof. **TOPROVE 7** □

Proof. **TOPROVE 8** □

6 Finding Claws Is almost Instance Optimal when the input has low Complexity

In this section we prove Theorem 1.7. That is, we prove Conjecture 1.6, that finding a claw in a graph is $O(\log(n))$ close to being instance optimal in the regime where finding a claw can be done in $O(\sqrt{n/\log n})$ queries by an algorithm with an unlabeled certificate. For what follows, let $\mathcal{P}_{S_3} = \mathcal{P}_{S_3}(n)$ denote the property of containing a claw (a 3-star) in n -vertex graphs.

Theorem 6.1 (Near-instance optimality in the low-complexity regime). *There exist universal constants $C, \alpha > 0$ and a Las Vegas algorithm $A_{\text{all-scales}}$ satisfying the following. For every graph G on n vertices in which the unlabeled certificate complexity satisfies*

$$\text{RAC}(\mathcal{P}_{S_3}, G) \leq \alpha \sqrt{\frac{n}{\log n}},$$

the algorithm $A_{\text{all-scales}}$ detects a claw with expected query complexity at most $C \log n \cdot \text{RAC}(\mathcal{P}_{S_3}, G)$.

The proof consists of three main parts, each contained in a separate subsection. Section 6.1 modifies the input graph to be more symmetrical to make all further analysis easier. Section 6.2 shows that it is not possible to find certain type of intersections (merges) between walks in time less than $\sqrt{n/\log(n)}$ unless one finds a claw. Lastly, in Section 6.3 we prove that conditioned on no two paths merging, a “memoryless” strategy that follows paths at logarithmically many different paths in parallel, where each path i is followed for length 2^i , is $O(\log n)$ -instance optimal.

6.1 Cleaning up the graph

We start by reducing the problem of finding a claw in an n -vertex graph G with n vertices to a closely related problem on a graph G' on the same vertex set, where G' additionally satisfies the following symmetry property, which will be useful for our analysis.

Definition 6.2 (path-symmetric graph). *A graph G is path-symmetric if for every vertex v of degree 1 in G , the other endpoint u of the simple path P containing v in G is also of degree 1.*

The construction. Given an undirected graph $G = (V, E)$ with $|V| = n$, V into disjoint sets $V_0 \cup V_1 \cup V_2 \cup V_{3+}$, where for $i = 0, 1, 2$, V_i contains all degree- i vertices in the graph, and V_{3+} includes all other vertices.

To each $v \in V_1 \cup V_2$ we can uniquely assign a path $P = P(v)$ that contains v . For any $v \in V_1$ (i.e., endpoint of the path $P(v)$), we say that v is a “yes” endpoint if the other endpoint u of $P(v)$ is in V_{3+} (i.e., u is a claw center); otherwise, v is a “no” endpoint (and in this case, $u \in V_1$ as well). Let V_1^{yes} and V_1^{no} denote the set of “yes” and “no” endpoints, respectively.

The transformation $G \mapsto G'$ is defined as follows. We add two new vertices w, w' and connect them to all vertices in V_1^{yes} within G . If $|V_1^{\text{yes}}| = 1$, we also connect w to w' . Thus, all vertices from V_1^{yes} are claw centers in G' . All other vertices in G remain unchanged. The degree of w, w' in G' depends on $n_{\text{yes}} = |V_1^{\text{yes}}|$; it is zero if $n_{\text{yes}} = 0$, two if $n_{\text{yes}} \in \{1, 2\}$, and three or more otherwise. Note that this transformation is deterministic. Thus, an algorithm holding an unlabeled certificate of G can easily construct from it an unlabeled certificate of G' .

It immediately follows from the construction that G' is path-symmetric. Our main lemma in this subsection states that finding claws in G' has roughly the same query complexity as in G .

Lemma 6.3. $\text{RAC}(\mathcal{P}_{S_3}, G) = \Theta(\text{RAC}(\mathcal{P}_{S_3}, G'))$, that is, the expected query complexity of finding a S_3 in G with an unlabeled instance is equal, up to a multiplicative constant, to the same query complexity for G' .

The same equivalence is true also without access to an unlabeled certificate.

The importance of Lemma 6.3 is that it allows us to eliminate from a graph all vertices of type V_1^{yes} . These vertices are problematic for our analysis; our proof in subsequent subsections relies heavily on symmetry arguments that break down when visiting a vertex of this type. The lemma implies that we can consider graphs in which each path P is of one of two types: either both endpoints of P are claw centers, or both are degree-1 vertices.

We note that Lemma 6.3 holds in any regime, not just the “low-complexity” one. In particular, the lemma may be useful toward proving Conjecture 1.6 in the general case.

Proof. **TOPROVE 11**

□

6.2 Hardness of merging walks

In this subsection we prove a hardness result concerning *merging walks* in graphs. We shall care about a property of *walks*, as *oriented subgraphs* of the input (undirected) graph. Although the main focus of this subsection is on the graph case, the definition below of a walk graph is suitable in both graphs and functions.

Definition 6.4 (Algorithm’s walk graph). *The walk graph of a query-based algorithm A in n -vertex graphs (in the query model) and functions $f: [n] \rightarrow [n]$ (treated as out directed graphs with out-degree 1) is initialized as an empty graph on $[n]$.*

Each time that an algorithm A queries a vertex v to obtain a neighbor u of it (or an out-neighbor, in the functions case), we add the oriented edge $v \rightarrow u$ to G_A .

A walk is any connected component (in the undirected sense) of the walk graph, preserving the orientations of the walk graph. That is, it is an object of the following structure:

$$v_{-k} \leftarrow v_{-k+1} \leftarrow \dots \leftarrow v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_l,$$

where $k, l \geq 0$ (and can equal zero), and v_0 is the first vertex to be queried among $\{v_i : -k \leq i \leq l\}$.

Note that the walk graph is kept oriented (each edge only has a single orientation); if $v \rightarrow u$ has already been added to G_A , then $u \rightarrow v$ cannot be added to it in the future.

Definition 6.5 (Merging walks). *Consider the walk graph G_A for an algorithm A in a graph G . We say that two walks*

$$v_{-k} \leftarrow \dots \leftarrow v_{-1} \leftarrow v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_l \quad \text{and} \quad u_{-k'} \leftarrow \dots \leftarrow u_{-1} \leftarrow u_0 \rightarrow u_1 \rightarrow \dots \rightarrow u_{l'}$$

in G_A with distinct starting points $v_0 \neq u_0$ merge if their endpoints intersect non-trivially:

$$\{v_{-k}, v_l\} \cap \{u_{-k'}, u_{l'}\} \neq \emptyset.$$

Our main result is a universal (not graph-specific) lower bound on the query complexity of finding merging paths, as long as a the walk-graph of the algorithm does not contain a claw center.

Lemma 6.6 (Hardness of merging for graphs). *There exists $\alpha > 0$ such that the following holds. For every n -vertex graph G and algorithm A making up to $q = \alpha\sqrt{n/\log n}$ queries to G , the probability that walks merge in G_A at least once during the run of A and that further G_A does not contain a claw center at the time the merging takes place, is at most $1/5$.*

The constant $1/5$ here is arbitrary (and can be replaced with any other small positive constant). The above lemma implies that the “strategy” of first finding merging walks and then using them to find a claw would require at least $\Omega(\sqrt{n/\log n})$ queries. In the next subsection, we show the complementary result: that in a graph where no merges have been observed, a simple strategy that does not require an unlabeled certificate is almost instance-optimal.

Toward the proof of Lemma 6.6. We start by introducing objects and ideas to be used in the proof of the lemma. We may assume that the input graph G does not contain claw centers, i.e., has maximum degree at most 2. (Indeed, querying any edge that contains a claw center makes the algorithm fail, and it is easy to show that removing all these edges cannot decrease the probability

to win.) Therefore, the graph G is a disjoint union of simple paths, simple cycles, and isolated vertices. For what follows, it will be convenient to measure the length of a path/cycle as the number of *vertices* (not edges) it contains, that is, an isolated vertex is a path of length 1.

We bucket the paths in G into sets of similarly-sized paths and cycles. Let $t = \log_{1.1} n = \Theta(\log n)$, and for every $0 \leq i < t$ let \mathcal{P}_i denote the collection of all paths and cycles in G containing at least 1.1^i and less than 1.1^{i+1} vertices. Further let N_i denote the sum of lengths of paths and cycles in \mathcal{P}_i . Note that these collections are disjoint and thus $\sum_{i=0}^{\log n} N_i = n$. Define

$$S = \left\{ 0 \leq i \leq t : N_i \geq \frac{\sqrt{n \log n}}{t} \right\},$$

where we note that $\sum_{i \notin S} N_i \leq \sqrt{n \log n}$. In particular, if an algorithm makes $q \leq \alpha \sqrt{n / \log n}$ queries then with probability $1 - 2\alpha$ none of the vertices in $\bigcup_{i \notin S} \mathcal{P}_i$ will be queried. That is, we may assume that the walk-graph of the algorithm does not intersect $\bigcup_{i \notin S} \mathcal{P}_i$, and contains only walks strictly in paths outside this union.

We prove the statement of Lemma 6.6 under an even stronger algorithmic model. Each time that a “new” vertex v is being queried by the algorithm A , we immediately notify the algorithm about which bucket \mathcal{P}_i the vertex v belongs to. Note that this implies, in particular, that at all times during the algorithm’s run, each walk W in the walk-graph of A is contained in a bucket known to the algorithm. We show that even under this stronger assumption, it is hard to merge walks.

Our first main claim is a concentration of measure argument regarding the number of walks within each bucket. It is used in the proof of Lemma 6.6 to show that walks in the same bucket have, very roughly speaking, a high “degree of freedom”, resulting in good bounds on the merging probability.

Claim 6.7 (Properties of walks). *There exists a constant $\alpha > 0$ satisfying the following. Let A be a query-based algorithm in G making $q \leq \alpha \sqrt{n / \log n}$ queries. The following statements hold with probability at least $9/10$.*

1. For all $0 \leq i < t$, the number of walks in G_A contained in $\bigcup_{P \in \mathcal{P}_i} P$ is bounded by $\frac{10N_i q t}{n}$.
2. For all $i \in S$ for which $1.1^i \leq \sqrt{\frac{n}{\log n}}$, the number of such walks is smaller than $\frac{1}{2} |\mathcal{P}_i|$.⁵

Proof. **TOPROVE 12** □

Proof. **TOPROVE 13** □

6.3 On algorithms for finding claws without merging

We next establish the near-instance-optimality of claw detection in the absence of merging between walks. Our main result is as follows.

Lemma 6.8 (Near instance-optimality in graphs without merges). *There exists a constant $\alpha > 0$ for which the following holds. Let G be a path-symmetric graph admitting an algorithm A for \mathcal{P}_{S_3} with expected query complexity $q_G \leq \frac{1}{20} \sqrt{n}$. Suppose that the probability of A to cause merging in G_A within its first $2q_G$ queries, provided that it has not found a claw before merging, is bounded by $1/5$. Then there exists an algorithm $A_{\text{all-scales}}$ without an unlabeled certificate, that finds a claw with expected query complexity $O(q_G \log n)$.*

⁵Note that the quantity $|\mathcal{P}_i|$ counts the *number* of paths in \mathcal{P}_i , not their total length.

- [HO20] Yi Hao and Alon Orlitsky. Data amplification: Instance-optimal property estimation. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, 2020.
- [HOSW18] Yi Hao, Alon Orlitsky, Ananda T. Suresh, and Yihong Wu. Data amplification: A unified and competitive approach to property estimation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, page 8848–8857, 2018.
- [HWZ21] Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, page 1166–1179, 2021.
- [KNY18] Ilan Komargodski, Moni Naor, and Eylon Yogev. Collision resistant hashing for paranooids: Dealing with multiple collisions. In *EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 162–194, 2018.
- [LM23] Alison Hsiang-Hsuan Liu and Nikhil S. Mande. Instance complexity of boolean functions. *CoRR*, abs/2309.15026, 2023.
- [MP91] Nimrod Megiddo and Christos H. Papadimitriou. On total functions, existence theorems and computational complexity. *Theor. Comput. Sci.*, 81(2):317–324, 1991.
- [Pap94] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *J. Comput. Syst. Sci.*, 48(3):498–532, 1994.
- [VV16] Gregory Valiant and Paul Valiant. Instance optimal learning of discrete distributions. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 142–155, 2016.
- [VV17] Gregory Valiant and Paul Valiant. An automatic inequality prover and instance optimal identity testing. *SIAM J. Comput.*, 46(1):429–455, 2017.