Interactive Graph Search

Yufei Tao
taoyf@cse.cuhk.edu.hk
Chinese University of Hong Kong
Hong Kong, China

Yuanbing Li
yb-li16@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Guoliang Li
liguoliang@tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

We study *interactive graph search* (IGS), with the conceptual objective of departing from the conventional "top-down" strategy in searching a poly-hierarchy, a.k.a. a decision graph. In IGS, a machine assists a human in looking for a target node z in an acyclic directed graph G, by repetitively asking questions. In each *question*, the machine picks a node u in G, asks a human "*is there a path from u to z?*", and takes a boolean answer from the human. The efficiency goal is to locate z with as few questions as possible. We describe algorithms that solve the problem by asking a provably small number of questions, and establish lower bounds indicating that the algorithms are optimal up to a small additive factor. An experimental evaluation is presented to demonstrate the usefulness of our solutions in real-world scenarios.

CCS CONCEPTS

• **Information systems** → **Collaborative search**.

KEYWORDS

Interactive Graph Search; Algorithms; Lower Bounds

ACM Reference Format:
Yufei Tao, Yuanbing Li, and Guoliang Li. 2019. Interactive Graph Search. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319885

1 INTRODUCTION

This paper considers a problem that we refer to as *interactive graph search* (IGS). It is concerned with the scenario where a human needs to explore a potentially massive poly-hierarchy — a.k.a. an acyclic directed graph (DAG) where each edge represents specialization — in order to locate the *deepest* node that best describes a certain concept. The DAG, typically, is stored at a remote server, and must be communicated to the human, with a unit cost charged on every node communicated. The algorithmic challenge is to devise a strategy to minimize the amount of interaction.

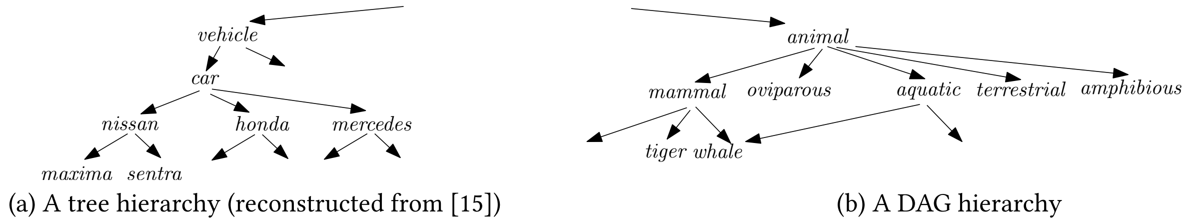
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319885

In Section 2, we will elaborate on the common patterns behind a class of applications that can be modeled as IGS, but for an immediate illustration here, let us examine a scenario from [15] where a machine summons a human's help to tag a picture according to a certain hierarchy. Figure 1a shows part of such a hierarchy, which is stored at the machine and is not known to the human. Interaction is initiated by the machine, which asks questions for the human to answer. Each *question* has the form: "*is this (picture) an* ***x***", where $\boldsymbol{x}$ is the name of a node. Here are some examples along a path in the hierarchy: "*is this a* ***car****?*", "*is this a* ***nissan****?*", and "*is this a* ***sentra****?*". Upon receiving a yes-answer to all of them, the machine can now place the tag *sentra* on the picture confidently.

The efficiency goal in the above scenario is to minimize the number of questions asked. More formally, one can think of the problem as a game between two players Alice and Bob. Initially, Bob secretly chooses a *target node* z in the hierarchy. Alice's job is to figure out which node is z. There is an *oracle* that Alice can inquire repeatedly. Each time, she picks (at her will) a *query* node q, and asks the oracle: *is there a (directed) path in the hierarchy from q to the node chosen by Bob?* Oracle reveals the answer (i.e., yes or no). So, what should be Alice's strategy in order to locate z with as few questions as possible?

For our example, one sees that the machine plays the role of Alice. The target node z is the final tag *sentra* that the machine should place on the picture. The human plays the role of oracle. Indeed, even though a human is not aware of the underlying hierarchy (let alone z), s/he can still correctly answer a question like "*is this a* ***car****?*" using her/his own knowledge and cognition power. A path exists in the hierarchy from $q = car$ to (the unknown) z if and only if the human answers yes.

While the above hierarchy is a tree, it can be a DAG in general. Figure 1b complements Figure 1a by showing another part of the hierarchy. When presented with the picture of a whale, a human will answer yes to both questions: "*is*



(a) A tree hierarchy (reconstructed from [15])

(b) A DAG hierarchy

Figure 1: Example hierarchies for human-assisted graph search

this a ***mammal***?" and "*is this an* ***aquatic*** *(animal)*?" This is consistent with the fact that both ***mammal*** and ***aquatic*** have paths leading to the node ***whale*** in the hierarchy. Indeed, to tag the picture correctly, the machine can reach the node ***whale*** by asking questions along either path.

Technical Challenges. The technical objectives of this work are two-fold:

- **(Upper bound)** Design an algorithm for Alice (i.e., the machine) to solve the problem with a small number of questions, regardless of Bob's choice of z.
- **(Lower bound)** Where is the limit of *all* algorithms? That is, how many questions must Alice ask in the worst case, no matter how smart she is?

Our problem is *online* in nature, namely, the next question Alice asks depends on the previous answers of the oracle. Such *interaction* with the oracle is essential for keeping the total number of questions small. Opposite to this is the *offline* version, as was studied by Parameswaran et al. in [15] under the name *human-assisted graph search*, where no interactions are permitted. Instead, Alice must ask all her questions *in one go*. Once the answers are returned, she must then do her best to figure out where is z. Suppose that Alice is constrained to ask no more than t questions for some small $t \geq 1$. In this case, she cannot guarantee finding z even with all the t answers collected; instead, all she can do is to narrow things down to a *candidate set* that must contain z. The objective (of the offline version) is to choose the t questions wisely to minimize the size of the candidate set.

To illustrate, consider Figure 1a with $t = 3$; and assume that Alice asks the oracle: "*is this an* $\boldsymbol{x}$?", where $\boldsymbol{x}$ = ***nissan***, ***honda***, and ***mercedes***, respectively. If the first question gets a yes answer, she is sure that z must be in {*nissan*, *maxima*, *sentra*}. This candidate set is her final knowledge because no more interaction with the oracle is allowed.

However, in the unlucky case where *none* of the questions returns yes, Alice has little information as to where is z. To see this, just imagine an overall hierarchy combining Figures 1a and 1b. Alice does not even have a clue whether z is a vehicle or an animal. Indeed, as shown in [15], the value of t must be rather large — often at the same order as the size of the hierarchy — to guarantee a small candidate set.

This issue goes away in IGS. As explained next, typically only a small number of questions suffices to locate z, even in the worst case.

Our Contributions. Let us now return to IGS. If the hierarchy has n nodes, the problem can be trivially solved with n questions: simply ask a question on every node. A bit less trivial is to do so with at most $d \cdot h$ questions — we will explain how in Section 2.2 — where d is the maximum out-degree of a node, and h is the length of the longest path in the hierarchy. Note that h is at least $\lceil \log_d n \rceil$, but can be as large as n when the hierarchy is a single path.

We show that the problem admits an algorithm with an alternative bound on the number of questions, and prove that the algorithm is nearly optimal:

- **(Upper bound)** We can find z in a DAG with at most $\lceil \log_2 h \rceil (1 + \lfloor \log_2 n \rfloor) + (d-1) \cdot \lceil \log_d n \rceil$ questions.
- **(Lower bound)** Any algorithm must ask at least $(d-1) \cdot \lfloor \log_d n \rfloor$ questions in the worst case. In other words, the proposed algorithm is optimal up to a small additive factor.

Our algorithm carefully decomposes the nodes of the input DAG hierarchy into disjoint subsets, where the nodes in each subset are connected by a path in the hierarchy. The decomposition allows us to navigate in the hierarchy through a series of binary searches on individual paths. This new technique is interesting in its own right, and is an outcome from the marriage of the *white-path theorem* and *heavy-path decomposition* (both will be explained in Section 3). In fact, the technique is — as we will show — powerful enough to settle near-optimally a more general variant of IGS where a human may need to answer multiple questions at a time.

Paper Organization. The rest of the paper is organized as follows. Section 2 will formally define the IGS problem, give a baseline solution, and present a class of applications that are adequately modeled by IGS. Section 3 reviews some preliminary techniques needed in our discussion. Sections 4 and 5 will present our algorithms and prove their theoretical guarantees, focusing on tree and DAG hierarchies, respectively. Section 6 will describe how to extend our algorithms to solve a more general variant of the problem. Section 7 will experimentally evaluate the performance of the proposed solutions using real data. Section 8 will survey the previous

work related to ours. Finally, Section 9 concludes the paper with a summary of findings.

2 INTERACTIVE GRAPH SEARCH

2.1 Problem Formulation

Next, we will formally define the *interactive graph search* (IGS) problem studied in this paper. Some of the notions that already appeared in Section 1 will be repeated for the reader's convenience.

We have a *hierarchy*, which is a connected DAG $G = (V, E)$. Define a node $v \in V$ as a *root* if it has an in-degree 0 (i.e., no incoming edges). We consider that G has only one root — if this is not true, simply add a dummy vertex to G with an outgoing edge to every original root. This dummy vertex has an out-degree equal to the number of roots in the original G, and now serves as the only root of G.

An adversary chooses arbitrarily a *target node* $z \in V$. An algorithm's goal is to identify which node is z. There is an *oracle* that can answer *questions*. Formally, in each question, the algorithm specifies a *query node* $q \in V$; and then the oracle returns a boolean answer denoted as *reach*(q):

- *yes*, if there is a (directed) path from q to z;
- *no*, otherwise.

In other words, the answer reveals the *reachability* from q to z, that is, *reach*(q) = *yes*, if and only if z is *reachable* from q, or equivalently, q can *reach* z. The algorithm is free to choose any query node in a question; and indeed, its choice in each question constitutes the core of the algorithm design.

The algorithm stops when it has figured out with no ambiguity where is z. Its *cost* is defined as the number of questions it has asked.

Throughout the paper, we set $n = |V|$, denote by d the maximum out-degree of the nodes in G, and by h the length of the longest path in G. For instance, if G is the DAG in Figure 2, then $n = 14$, $d = 3$ and $h = 5$ (the path from node 1 to node 14 is the longest in G).

2.2 A Baseline Top-Down Solution

IGS has the following simple *out-neighbor property*:

Proposition 1 (Out-Neighbor Property). *Suppose that we already know reach*(u) = *yes for some node* $u \in V$. *Then:*

- $u = z$ *if and only if every out-neighbor* v *of* u *satisfies reach*(v) = *no;*
- $u \neq z$ *if and only if* u *has an out-neighbor* v *satisfying reach*(v) = *yes.*

Example. To illustrate, suppose that the input DAG G is the graph in Figure 2. Assume that the target node z is node 8. Set u to z; the first bullet says that no out-neighbors of u can reach z (that is rather trivial). Set instead u to node 2; it is clear that *reach*(node 2) = *yes*. The second bullet says that node 2 must have an out-neighbor that can reach z. Indeed, in this case, both nodes 3 and 4 can be this out-neighbor. □

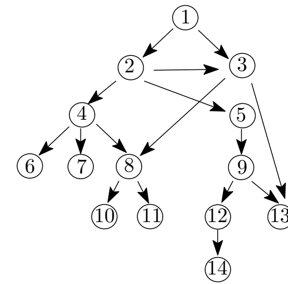


Figure 2: A DAG hierarchy

The property motivates a straightforward `top-down` algorithm for IGS. At the beginning, set u to the root of G. At each step, query the oracle on every out-neighbor of u, until finding an out-neighbor v with *reach*(v) = *yes*. If no such v exists, we terminate by returning u as the target node. Otherwise, we set u to v, and repeat.

Clearly, `top-down` asks at most d questions at every u. By moving from u to v, it walks a step along a path in G. Hence, the algorithm asks at most $d \cdot h$ questions in total.

2.3 Applications

The IGS problem offers an algorithmic framework for studying how to minimize *interaction* in applications where the objective is to locate the most specific node in a *decision tree* — or its extension *decision graph* [13] — that best fulfills an information need. Figure 1a illustrates a decision tree, while Figure 1b exemplifies a decision graph, which can be regarded as a decision tree but with identical subtrees merged. Conventionally, the top-down strategy in Section 2.2 has been the norm for exploring a decision tree/graph. Philosophically, the goal of IGS is to seek a way to beat that conventional wisdom.

The concrete scenarios for such applications are versatile. What is described in Section 1 is known as *image categorization* in [15]. Next, we will describe several other applications. Our selection strives to achieve diversity: each application below is representative with distinct features.

Manual Curation. In the example with Figures 1a and 1b, the input hierarchy is *fixed*, with the goal being to find a node to fit a certain object (i.e., a picture). In manual curation, on the other hand, we want to *extend* a hierarchy by inserting a new node $\boldsymbol{x}$, e.g., a new brand of ***nissan***. This is effectively an instance of IGS, whose output is the node that should parent $\boldsymbol{x}$. In reality, many hierarchies (better known as taxonomies or categories) require this kind of periodic extensions; some examples are Wikipedia, web of concepts, ACM computing classification system, and so on.

Relational Databases. Often times a user may need to search a database without being aware of the table schemata, ruling out the possibility to write an accurate SQL query to fetch the information targeted. This motivated *faceted search* [17, 20], where the system interacts with the user by asking increasingly refined questions that eventually lead to the data to be retrieved. These questions are selected during preprocessing, and are organized into a decision tree/graph, after which faceted search can be performed online by descending an appropriate path in the tree/graph. Our IGS algorithms nicely complement faceted search, which is exactly an instance of IGS.

A Commercial Site. *Zingtree.com* is the portal site of a company that specializes in helping organizations build a sophisticated decision tree/graph designed to facilitate one of the following services: *technical support*, *call centers*, *customer care*, *retail*, and *medical and health*. To provide, for example, technical support, an organization would rely on the decision tree/graph to interact with a customer, in order to diagnose the problem encountered by the customer and to suggest the corresponding remedy. This is a typical scenario of IGS. The algorithms in this paper can be integrated with any of those decision trees/graphs to reduce the amount of interaction demanded (which is crucial for the services aforementioned).

3 PRELIMINARIES

3.1 Heavy-Path Decomposition

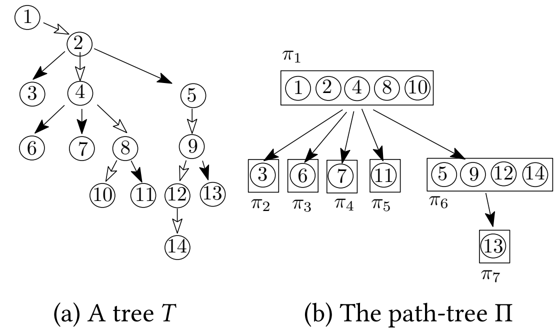
In this subsection, we give a self-contained tutorial to the *heavy-path decomposition* technique [18]. Let T be a tree of n nodes which may *not* be balanced, i.e., its height can be arbitrarily close to n. The goal of heavy-path decomposition is to produce a balanced representation of T.

We need to be first familiar with the notions of "heavy edges" and "light edges". Consider u to be an internal node in T. Let v be the child node of u whose subtree has the largest size[1] (ties broken arbitrarily). The edge between u and v is said to be *heavy*, while the other out-going edges of u are said to be *light*.

Example. Suppose that T is the tree in Figure 3a (all the edges are pointing downwards). The subtree of node 4 has a size 6, while that of node 5 has a size 5. Set u to node 2. Among its three out-going edges, the one pointing to node 4 is heavy, while the other two are light — because node 4 is the child of u with the largest subtree. In the figure, all the heavy edges are represented using white arrows, whereas the light ones have black arrows. □

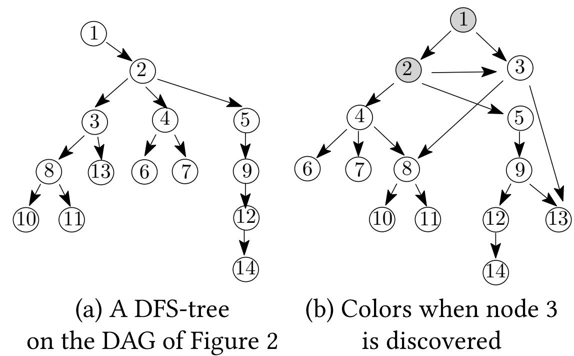
Now, concatenate heavy edges into *maximal* paths, i.e., no path can be extended with yet another heavy edge. Every resulting path is called a *heavy path*.

[1]The size of a subtree is the number of nodes therein.



(a) A tree T (b) The path-tree Π

Figure 3: Heavy-path decomposition



(a) A DFS-tree on the DAG of Figure 2 (b) Colors when node 3 is discovered

Figure 4: White-path theorem

Example (cont.). In Figure 3a, path "nodes 2 → 4 → 8", which we abbreviate as (2, 4, 8) henceforth, is not a heavy path, because it can be extended with a white edge either in front or at the end. On the other hand, path (1, 2, 4, 8, 10) is a heavy path, and so is (5, 9, 12, 14). Do not forget there are five more heavy paths: (3), (6), (7), (11), and (13); they are heavy paths of length 0. □

Every node appears in *one and exactly one* heavy path (observe this property from Figure 3a.) By viewing each heavy path as a whole, we can define a *path tree* Π as follows:

- Treat each heavy path as a "super-node", and make it a vertex in Π.
- Given two heavy paths $\hat{\pi}$ and π, add an edge in Π from $\hat{\pi}$ to π if and only if a node of $\hat{\pi}$ parents the first node of π in T.

Example (cont.). Figure 3b shows the path tree Π for the T in Figure 3a. Π has 7 vertices $\pi_1, \pi_2, ..., \pi_7$, corresponding to 7 heavy paths in T, respectively. There is an edge from π_1 to π_6 because node 2 of π_1 parents node 5 — the first node of π_6 — in T. Likewise, an edge exists from π_6 to π_7 because node 9 parents node 13 in T. □

The path tree Π in Figure 3b has 3 levels. It can be proved that Π cannot be too tall in general:

Lemma 1 ([18]). *Π has at most* $1 + \lfloor \log_2 n \rfloor$ *levels.*

3.2 DFS and White-Path Theorem

We devote this subsection to *depth-first search* (DFS), which will play an essential role in our IGS solutions. The DFS algorithm is "deceptively simple", and is endowed with numerous interesting properties. Our main goal is to review the *white-path theorem*: the famous theorem that explains why DFS is the key to solving a long list of non-trivial problems, e.g., cycle detection, topological sort, finding strongly-connected components, etc. The IGS problem will be a new member on the list, as we will show in this paper.

DFS. We will only be concerned with a connected DAG $G = (V, E)$ that has a single root r. DFS traverses G by resorting to a stack and a vertex coloring scheme:

- **White**: a vertex has never been pushed into the stack.
- **Gray**: a vertex is currently in the stack.
- **Black**: a vertex has been popped out of the stack.

In the outset, the stack contains only r (we always start DFS from the root in this work). Accordingly, all the vertices are colored white, except r, which is colored gray. The algorithm then proceeds as follows:

```
1.  while stack not empty
2.      u ← the vertex at the top of the stack
3.      if u has any white out-neighbor v
4.          push v into the stack, and color it gray
        else
5.          pop u out of the stack, and color it black
```

At the moment right before v turns gray at Line 4 — namely, after it is found at Line 3 as a white out-neighbor of u — we say that v is *discovered*, and u is its *finder*. Every node other than r is discovered once and exactly once in the algorithm.

Example. Let G be the DAG in Figure 2. Suppose that, at Line 3, we adopt the policy that the out-neighbors of u be picked in ascending order of node id. At the beginning, the stack has only node 1. Node 2 is discovered next, with node 1 as the finder. In turn, node 2 is the finder for node 3, which is the finder for node 8, which is the finder of node 10. At this moment, the stack has (from bottom to top): nodes 1, 2, 3, 8, 10; these five nodes are in gray, while the other nodes are still white. Node 10 is then popped out, and turns black. Node 8 currently tops the stack, with only one white out-neighbor: node 11. Hence, node 11 is discovered next, making the stack: nodes 1, 2, 3, 8, 11. We omit the rest of the execution. □

DFS-Tree. The traversal order of DFS defines a *DFS-tree* T as follows:

- The set of vertices of T is just V.
- T is rooted at r.
- If a node u is the finder of a node v, u parents v in T.

Example (cont.). Figure 4a gives the DFS-tree for the execution of DFS illustrated earlier. It is worth mentioning that every path emanating from the root represents the content of the stack at some point of the algorithm. For example, the path $(1, 2, 3, 8, 11)$ is the content of the stack right after node 11 was discovered (as shown earlier). □

Based on the DFS-tree T, every edge $(u, v) \in E$ can be classified into one of the three categories below:

- **Tree edge**: u is the parent of v in T.
- **Forward edge**: u is a proper ancestor of v in T.
- **Cross edge**: neither u nor v is an ancestor of the other.

Example (cont.). In Figure 2 (which is reproduced in Figure 4b), edge $(1, 3)$ is a forward edge with respect to the DFS-tree of Figure 4a, $(4, 8)$, $(9, 13)$ are cross edges, while the other edges are tree edges. □

White-Path Theorem. The theorem points out a crucial property of DFS. Consider the moment when a node u is just discovered (it is about to be pushed into the stack). Suppose that there is a *white path* — namely a path where all the vertices are white — starting from u and ending at another vertex v. In other words, the vertices on this path have not been discovered yet. It is guaranteed that the algorithm must be able to discover v while u is still in the stack.

Example (cont.). Figure 4b shows the color state when node 3 is discovered in our earlier execution of DFS. At this moment, node 3 has white paths to nodes 8, 10, 11, 13. Then, for sure, before node 13 turns black (i.e., while it still remains in the stack), DFS will definitely have discovered all those 4 nodes. □

The white-path theorem states the above property formally by resorting to the DFS-tree.

Theorem 1 (White-Path Theorem [1]). *In the DFS-tree, a node u is a proper ancestor of a node v if and only if the following is true: when u is discovered, there is a white path from u to v.*

Example (cont.). Indeed, nodes 8, 10, 11, 13 are the only proper descendants of node 3 in the DFS-tree of Figure 4a. □

4 ALGORITHMS FOR TREES

Let us "warm up" by dealing with a special version of IGS. Recall that the underlying hierarchy is a DAG G. In this section, we will focus on the case where G is a *tree*. This allows us to present some of our techniques (particularly, those related to heavy-path decomposition) without the other details needed to cope with DAGs. Since we will be concerned only with a tree hierarchy, we will denote the hierarchy as T (rather than G). Let r be the root of T. Every edge in T is directed away from r. As defined in Section 2.1, we denote by d the maximum out-degree of a node in T, and by h the length of the longest (directed) path.

4.1 The First Algorithm

In the extreme case where T is a single path of length h, it is trivial to find the target node z by binary search in at most $\lceil \log_2 h \rceil$ questions. What makes binary search work is monotonicity. In general, on any directed path π, we always have two monotone properties:

- If $reach(u) = yes$ for a node u on π, then $reach(v)$ must also be *yes* for any node v *before* u on π.
- If $reach(u) = no$ for a node u on π, then $reach(v)$ must also be *no* for any node v *after* u on π.

How to exploit the monotonicity on a general tree hierarchy T? This is where heavy-tree path decomposition comes in. First, perform such a decomposition on T, and obtain a path tree Π, in the way introduced in Section 3.1. Then, we can carry out the search by *interleaving* between T and Π. The algorithm, named `interleave`, is formally described as follows:

algorithm `interleave`
1. $\pi \leftarrow$ the root (super-node) of Π /* π is a path in T */
2. **repeat**
3. /* navigate in Π */
   binary search π to find the last node u
   with $reach(u) = yes$
4. /* navigate in T */
   find a child v of u in T with $reach(v) = yes$
   (note that v cannot be in π)
5. **if** v does not exist **then return** u
   else
6. $\pi \leftarrow$ the (only) super-node in Π containing v
   /* π is a path in T, and v must be the first node
   in this path */

Example. To illustrate the algorithm, assume that T is the tree in Figure 3a, whose decomposition tree Π is in Figure 3b. Suppose that the adversary has secretly decided the target node z to be node 9.

`Interleave` starts by looking at the root π_1 of Π. At Line 3, it performs binary search on π_1 to find node 2, which is the last node on π_1 that can reach z. Then, the algorithm jumps to node 2 in T, and examines its child nodes 3 and 5 (child node 4 can be left out because it is in π_1, and hence, has already been considered in the binary search on π_1). After issuing a question on each node, we find that $reach(\text{node } 5) = yes$; thus, v = node 5 at Line 4. At Line 6, `Interleave` switches back to Π, and identifies the path π_6, i.e., the super-node in Π covering node 5.

Continuing, `Interleave` (at Line 3) performs binary search on π_6, which finds node 9 as the last node in π_6 that can reach z (note: at this moment, the algorithm still does not know that z is just node 9). At Line 4, it turns back to T to inspect the child node 13 of node 9 (child node 12 can be left out). As $reach(\text{node } 13) = no$, now we can conclude that z is node 9. The algorithm finishes here. □

Next, we analyze the number of questions asked by `interleave` in the worst case. Call Lines 3-7 an *iteration*. Since π has a length of at most h, the binary search at Line 3 requires at most $\lceil \log_2 h \rceil$ questions. Line 4 obviously requires no more than d questions because u can have at most d child nodes. This caps the number of questions per each iteration at $d + \lceil \log_2 h \rceil$.

How many iterations are needed? The crucial observation is that, every time we come to Line 4, we have descended one level of Π. By Lemma 1, Π has at most $1 + \lfloor \log_2 n \rfloor$ levels. Hence, the number of questions is $O(\log n \cdot \log h + d \cdot \log n)$. We do not need to be bothered by the hidden constants here because the result will be improved very shortly.

4.2 Improving the Cost

Next, we reduce the number of questions of `interleave` by making a small modification to the algorithm.

At Line 4, `interleave` finds a child node v of u in T with $reach(v) = yes$. We did so with d questions by querying the children of u in an arbitrary order. Now, we apply a particular ordering:

> query the child nodes v of u — but ignoring
> the child node in π — in non-ascending order
> of the subtree size of v.

As soon as a child node returns $reach(v) = yes$, we stop and proceed to Line 5. We refer to the modified algorithm as `ordered-interleave`.

Example. For an illustration, consider again the execution of `interleave` traced out earlier. Recall the moment after node 2 is found in π_1 through binary search. As explained before, it suffices to consider child nodes 3 and 5 of node 2 (because node 4 is in π_1). While `interleave` inspects the child nodes in an arbitrary order, `ordered-interleave` processes node 5 first, because it has a larger subtree than node 3. □

The modification provably reduces the worst-case cost:

LEMMA 2. `Ordered-interleave` *asks at most* $\lceil \log_2 h \rceil \cdot (1 + \lfloor \log_2 n \rfloor) + (d-1) \cdot \lceil \log_d n \rceil$ *questions.*

PROOF. See Appendix A. □

4.3 A Lower Bound

We finish the section by proving a lower bound on the number of questions needed to perform IGS on a tree hierarchy.

LEMMA 3. *Given a tree hierarchy, any algorithm must ask at least* $(d-1) \cdot \lfloor \log_d n \rfloor$ *questions in the worst case.*

PROOF. See Appendix B. □

The lower bound matches the upper bound in Lemma 2 up to a small additive factor.

5 ALGORITHMS FOR DAGS

We are now ready to attack the IGS problem in its general form: the input hierarchy is a DAG $G = (V, E)$. Our algorithm, in essence, reduces the problem to that on a special DFS-tree of G — which we name the *heavy-path DFS-tree* — that integrates features of both DFS-tree and heavy-path decomposition.

5.1 The Heavy-Path DFS-Tree

Consider performing DFS on G starting from the root. Recall that, at each step, the algorithm takes the top node u of the stack, and looks for a white out-neighbor v of u to visit next (as shown at Line 3 of the pseudocode in Section 3.2). Normally, any out-neighbor v would suffice. We, however, will insist on choosing the most "out-reaching" v.

Formally, let S be the set of white out-neighbors of u at this moment. For each $v \in S$, we count the number — denoted as $count(v)$ — of nodes that v can reach via white paths. Then, the node v to visit next is the one in S with the largest $count(v)$, breaking ties arbitrarily.

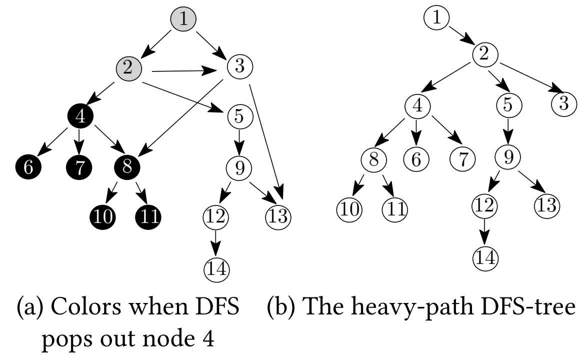
Example. To illustrate this, let us consider Figure 4b again. Remember that, at this moment, the stack contains (from bottom to top): nodes 1 and 2. These two nodes are gray, while the other nodes are still white. Since node 2 tops the stack, we need to decide which of its white out-neighbors should be visited next. From Figure 4b, one can see that node 3 can reach five nodes via white paths at this moment: nodes 3, 8, 10, 11, 13; hence, $count(\text{node } 3) = 5$. On the other hand, $count(\text{node } 4) = 6$ because node 4 can reach six nodes via white paths: nodes 4, 6, 7, 8, 10, 11. Finally, $count(\text{node } 5)$ is also 5. Therefore, the node to visit next is node 4.

As another illustration, Figure 5a shows the color state when node 4 is popped out of the stack. At this moment, the stack once again contains (bottom to top): nodes 1 and 2, which are in gray. Nodes 4, 6, 7, 8, 10, and 11 are currently black (they have been pushed and then popped from the stack). Which out-neighbor of node 2 to pick this time? Now, node 2 has only two white out-neighbors: nodes 3 and 5. Notice that $count(\text{node } 3)$ has decreased to 2: node 3 can reach only itself and node 13 via white paths. Since $count(\text{node } 5) = 4$, next DFS visits node 5. □

Define T as the DFS-tree corresponding to running DFS in the way explained above. We "regulate" T by arranging its nodes as follows:

> At each internal node u of T, arrange its child nodes from left to right in the order that those child nodes are discovered.

The resulting T has a nice property: a pre-order traversal of T enumerates the nodes in the same order as they were discovered in DFS.



(a) Colors when DFS pops out node 4 (b) The heavy-path DFS-tree

Figure 5: Computing the heavy-path DFS-tree

Example (cont.). Using the ordering strategy introduced earlier, DFS discovers the nodes in this sequence: 1, 2, 4, 8, 10, 11, 6, 7, 5, 9, 12, 14, 13, 3. Figure 5b shows the corresponding DFS-tree. Note that a pre-order traversal of the tree gives precisely the same node sequence. □

The lemma below explains why we refer to this T as the *heavy-path DFS-tree*:

LEMMA 4. *Consider any internal node u. Let v_1, v_2 be child nodes of u such that v_1 is on the left of v_2. Then, the subtree of v_1 in T is at least as large as that of v_2.*

PROOF. See Appendix C. □

Example (cont.). Observe that, in Figure 5b, the child nodes of each internal node have been automatically ordered from left to right in non-ascending order of subtree size. □

5.2 The Algorithm

Our algorithm for performing IGS on a DAG — named `DFS-interleave` — can be formally described as:

algorithm `DFS-interleave`
/* T is the heavy-path DFS-tree */
1. $\hat{u} \leftarrow$ the root r
2. **repeat**
   /* invariant: z is reachable from $\hat{u}$ */
3. $\pi \leftarrow$ the leftmost $\hat{u}$-to-leaf path of T
4. binary search π to find the last node u with $reach(u) = yes$
5. find the leftmost child v of u in T with $reach(v) = yes$
   /* note that v cannot be in π */
6. **if** v does not exist **then return** u
7. **else** $\hat{u} \leftarrow v$

Example. We illustrate the algorithm by setting G to the graph in Figure 5a, whose heavy-path DFS-tree T is shown in Figure 5b. Suppose that the adversary has secretly chosen the target node z to be node 9.

DFS-interleave first identifies at Line 3 the leftmost root-to-leaf path of T: $\pi = (1, 2, 4, 8, 10)$. The algorithm performs binary search on π to find node 2, which is the last node on π that can reach z. At Line 4, we find node 5, which is the leftmost child of node 2 that can reach z — this requires only one question: since node 5 is on the left of node 3, we test *reach*(node 5) first, which turns out to be *yes*, and thus, removes the need to test node 3 (note: node 4 does not need to be considered because it is on π).

Now the execution goes back to Line 3, where we set $\pi = (5, 9, 12, 14)$. The binary search at Line 4 finds node 9. Then, we test *reach*(node 13), which is *no*. The algorithm terminates here by returning node 9. □

The correctness proof of DFS-interleave is somewhat technical, and can be found in Appendix D.

5.3 Analysis

Interestingly, the cost analysis of DFS-interleave is completely the same as that of ordered-interleave. To see why, let us first observe:

COROLLARY 1. *Consider any internal node u of T, and a child node v of u in T. The edge (u, v) is heavy if and only if v is the leftmost child of u.*

PROOF. Immediately from Lemma 4. □

The next lemma may come as a pleasant surprise:

COROLLARY 2. *The path π identified at Line 3 (in any iteration) must be a heavy path in T.*

PROOF. Immediately from Corollary 1 and the definition of heavy path. □

Imagine that we perform a path-decomposition of T to obtain its corresponding path tree Π. Equipped with Corollary 2, it is easy to verify that DFS-interleave asks exactly the same questions as running ordered-interleave on Π. Therefore, the upper bound in Lemma 2 holds directly on DFS-interleave.

Example (cont.). Let us look at the example shown in Figure 5 one more time. One can verify that a heavy-path decomposition of the tree in Figure 5b gives precisely the path tree in Figure 3b. Indeed, the running of DFS-interleave follows the same steps as running ordered-interleave on Figure 3b. □

Finally, the lower bound in Lemma 3 still holds on general DAG hierarchies because a tree is a DAG. With this, we have arrived at the first main result of this paper.

THEOREM 2. *Both the following statements are true about the IGS problem:*

- DFS-interleave *asks at most* $\lceil \log_2 h \rceil \cdot (1 + \lfloor \log_2 n \rfloor) + (d-1) \cdot \lceil \log_d n \rceil$ *questions.*
- *Any algorithm must ask at least* $(d-1) \cdot \lfloor \log_d n \rfloor$ *questions in the worst case.*

6 EXTENSIONS

In our IGS problem so far, in each question, we can ask the oracle to resolve the reachability of only one node. In practice, an algorithm may invite a human to resolve the reachability of multiple nodes at a time. Next, we show that our algorithms can be extended easily to these scenarios.

The k-IGS Problem. Let us start by extending the problem definition. As before, we have a DAG hierarchy $G = (V, E)$ with a single root; and an adversary secretly chooses a target node z. An algorithm's goal is still to find z by resorting to an oracle.

The oracle, however, is k-times more powerful, where $k \geq 1$ is an integer. Formally, in a *k-question*, a query specifies a *query set* Q of nodes $q_1, q_2, ..., q_k$ in V. The oracle returns k boolean values $b_1, b_2, ..., b_k$, where $b_i = 1$ ($i \in [1, k]$) if z is reachable from q_i (i.e., *reach*(q_i) = *yes*), and $b_i = 0$, otherwise. Accordingly, the *cost* of the algorithm is defined as the number of k-questions issued. We refer to this problem as k-IGS, which captures IGS as a special case with $k = 1$.

Top-Down. The out-neighbor property in Proposition 1 still holds. Recall that the top-down algorithm works by repeating the following step: given a node u with *reach*(u) = *yes*, find an out-neighbor v of u with *reach*(v) = *yes* (if v exists). In IGS, this step required cost d, where d is the largest number of out-neighbors that u may have. In k-IGS, the step can be implemented using $\lceil d/k \rceil$ k-questions, where each question includes k out-neighbors of u in the query set. Therefore, top-down entails a cost of $\lceil d/k \rceil \cdot h$, where h is the length of the longest path in G.

Ordered-Interleave. Next, we will show how the proposed algorithms can be adapted to k-IGS, starting with ordered-interleave designed for tree hierarchies.

The algorithm still runs in the way described by the pseudocode in Section 4.1, with Line 4 implemented using the ordering idea of Section 4.2. There are only two differences:

- At Line 3, the binary search is replaced by k-ary search (which works in a fashion similar to searching for a key in a B-tree). Specifically, suppose that the path π is the sequence of nodes $u_1, u_2, ..., u_x$; the objective is to find the largest $y \in [1, x]$ such that *reach*(u_y) = *yes*. We inquire the oracle with a query set $Q = \{u_{x/k}, u_{2x/k}, ..., u_x\}$. If i is the largest integer satisfying *reach*($u_{i \cdot x/k}$) = *yes*, we know that y must be $[i \cdot x/k, (i+1)x/k)$, which is then searched recursively. For simplicity, we have assumed x to be a multiple of k, because it is trivial to extend the strategy to arbitrary

x, so that y can be determined in $\lceil \log_k x \rceil \le \lceil \log_k h \rceil$ k-questions.
- At Line 4, we query k child nodes of u in one k-question.

Lemma 5. `Ordered-interleave` *with the above adaptation makes at most* $(1 + \lceil \log_k h \rceil)(1 + \lfloor \log_2 n \rfloor) + \frac{d-1}{k} \lceil \log_d n \rceil$ *k-questions to solve k-IGS on a tree hierarchy.*

Proof. See Appendix E. □

DFS-Interleave. Consider now k-IGS on a general DAG hierarchy G. Section 5 proved that `DFS-interleave` can be regarded as running `ordered-interleave` on the heavy-path DFS-tree of G. The same proof holds here. In other words, `DFS-interleave` settles k-IGS with at most the cost given in Lemma 5.

In general, *any* lower bound L on the worst-case cost of IGS immediately implies a lower bound L/k on k-IGS. This is because any k-IGS algorithm A with cost U implies an IGS algorithm with cost $U \cdot k$, by implementing each k-question issued by A with k questions to the oracle of IGS. The above discussion brings us to our general result:

Theorem 3. *Both the following statements are true about the k-IGS problem:*
- `DFS-interleave` *asks at most* $(1 + \lceil \log_k h \rceil)(1 + \lfloor \log_2 n \rfloor) + \frac{d-1}{k} \cdot \lceil \log_d n \rceil$ *k-questions.*
- *Any algorithm must ask at least* $\frac{d-1}{k} \cdot \lfloor \log_d n \rfloor$ *k-questions in the worst case.*

7 EXPERIMENTS

7.1 IGS under Reliable Oracles

The experiments of this subsection were designed to study the characteristics of IGS algorithms under the algorithmic framework proposed in Section 2.1. In particular, the oracle is *reliable*, i.e., it does not make mistakes. Applications with such oracles are the primary beneficiaries of our solutions.

Data. We deployed two datasets:
- `Amazon`: this is a tree that represents the product hierarchy at Amazon. The tree was obtained from the file *metadata.json.gz* downloadable at *jmcauley.ucsd.edu/data/amazon/links.html* [7]. The file contains a record for each product sold at Amazon. The record has a field named *categories*, which specifies the nodes on the path from the root to the product's category. For example, here is what the path for a book on US history looks like: [∗, Books, History, Americas, United States] (where ∗ means the root). We reconstructed the product hierarchy as the trie on all these paths. The resulting tree has 29,240 nodes.

depth	average out-degree
0	84
1	11
2	4.6
3	2.4
4	0.97
5	0.33
6	0.17
7	0.13
8	0.11
9	0

(a) `Amazon`

depth	average out-degree
0	8
1	83
2	3.4
3	2.2
4	1.4
5	0.87
6	0.71
7	0.59
8	0.54
9	0.48
10	0.69
11	0.44
12	0

(b) `ImageNet`

Table 1: Out-degree statistics

- `ImageNet`: this is a DAG that represents the organization of a collection of images according to WordNet. The DAG was obtained from *www.image-net.org/api/xml/structure_released.xml*. Each *synset* tag in the XML document represents a node, whose id is given in the *wnid* attribute of the tag. The out-neighbors of the node are explicitly given inside the tag. We retained all the nodes, except the one with *wnid* = "fa11misc" because this node contains miscellaneous images that do not conform to WordNet. The DAG has 27,714 nodes.

For each dataset, Table 1 shows the average out-degree of the nodes at each *depth*. Recall that, in general, the *depth* of a node in a DAG (with a single root) is the length of the shortest path from the root to the node (this definition applies to a tree as well). It is clear from the table that, for both datasets, nodes closer to the root tend to have more out-neighbors. Note that an average out-degree can be less than 1 at a depth where there are many leaves (a *leaf* in a DAG is a node with out-degree 0).

Competing Algorithms. Our objective is to evaluate the usefulness of the proposed `DFS-interleave` algorithm, using the `top-down` algorithm (see Section 3) as a benchmark. As explained in Section 5.3, on a tree, `DFS-interleave` degenerates into the `ordered-interleave` algorithm in Section 4.2. So one can think conveniently that the competition was between `ordered-interleave` and `top-down` on `Amazon`, but between `DFS-interleave` and `top-down` on `ImageNet`. We left out the `interleave` algorithm in Section 4.1 because it served as a stepping stone towards `ordered-interleave`.

A remark about `top-down` is in order. Recall that at each node u, the algorithm examines its out-neighbors v in turn, until finding the first one with $reach(v) = 1$. This, however, means that the algorithm's performance is highly sensitive to how the out-neighbors are ordered. To avoid "pinning" the algorithm to any particular ordering, we adopted the implementation that the out-neighbors of u were examined

based on a random permutation. As such, `top-down` became a randomized algorithm. Every measurement reported in our experiments was averaged from 10 runs of this algorithm.

Workload. Recall that, in IGS or k-IGS, an adversary specifies a target node. Different target nodes define different *instances* of the problem. We considered all the possible instances defined by every single leaf in the underlying hierarchy. All these instances together constituted a *workload*. The workloads on `Amazon` and `ImageNet` had 24,329 and 21,427 instances, respectively (these are the number of leaves in each dataset).

Metrics. The primary metric for assessing an algorithm was its *cost*, i.e., number of questions or k-questions issued.

We also used another metric — *candidate set size* (CSS) — to measure an algorithm's progressiveness. Specifically, at any moment during an algorithm's execution, the CSS is the number of leaf nodes that the algorithm still can*not* rule out (i.e., every such a leaf could still be the target node). For sure, the CSS monotonically decreases as the algorithm runs; and the algorithm cannot stop until the CSS has dropped to 1. Ideally, we would like the algorithm to reduce CSS substantially with just a few questions/k-questions. Indeed, in practice, one may even choose to terminate an algorithm manually once the CSS has become sufficiently small.

Machine and Coding. In all the experiments, CPU computation was carried out on a machine equipped with an Intel Core i7-4870HQ CPU at 2.5GHz, and 16 GB of memory. All our implementations were programed in Python.

Results on IGS. Let us start with `Amazon`. The first experiment aims to evaluate the efficiency of `ordered-interleave` and `top-down` when the target node was placed at various depths. For this purpose, we used each algorithm to run a workload. For each depth value d, we calculated the average cost of the algorithm on all the instances defined by the leaves of depth d. The results are presented in Figure 6a.

When $d = 1$ (namely the target node is directly below the root), `top-down` was better because in this case the binary searches performed by `ordered-interleave` offer little help, and thus, do not pay off. However, `ordered-interleave` started to outperform `top-down` as soon as d increased to 2; and the gap between the two algorithms was fairly significant for all the other depth values. In general, the binary searches of `ordered-interleave` are more effective when the target node lies deeper in the tree — because a single binary search can skip multiple levels, which would need to be "plowed through" by `top-down`.

It is worth pointing out that the cost of `top-down` does *not* need to grow with the depth — note the "surge" in its cost at $d = 3$ and the "dip" at $d = 4$. In general, this algorithm is sensitive to how many children are owned by the nodes on the path from the root to the target leaf (we will delve into this issue later in Figure 6c). Indeed, in `Amazon`, many depth-3 leaves gather under large-fanout ancestors that do not have leaves of depth 4 or more. This is the reason behind the aforementioned surge and dip.

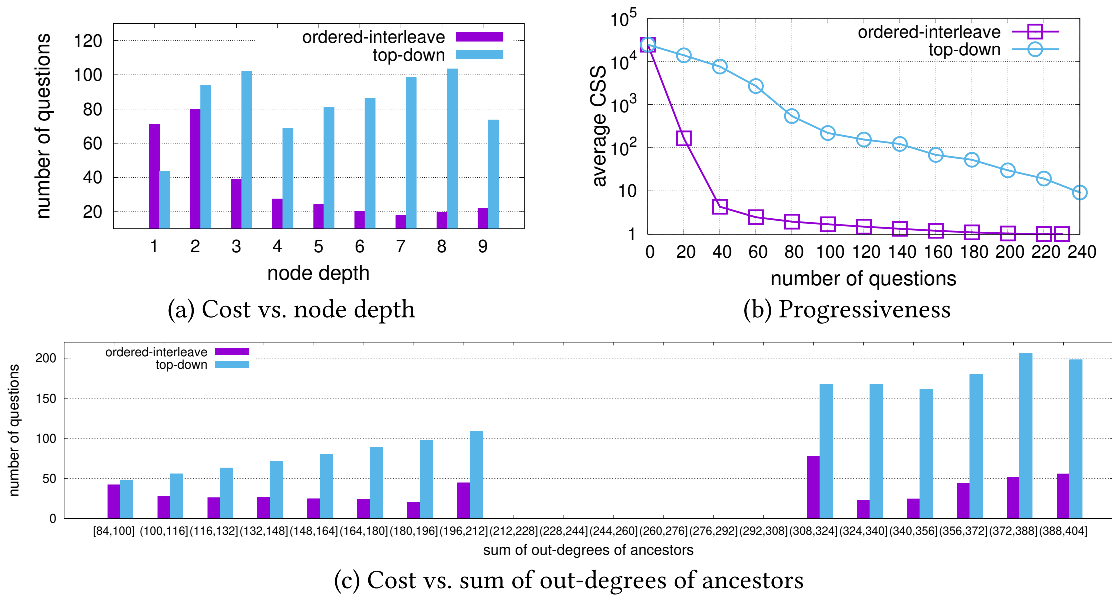
To demonstrate the progressiveness of each algorithm, we designed an experiment as follows. For each instance in a workload, we generated an array CSS that had (conceptually) an infinite length, such that $CSS[i]$ ($i \geq 1$) was set to the CSS at the moment right after the algorithm had entailed a cost of i. After the algorithm had terminated at some cost — say c — we set $CSS[i] = 1$ for every $i \geq c$. In this way, $CSS[i]$ bore an intuitive meaning: *a cost budget of* $i \geq 1$ *guaranteed a CSS equal to* $CSS[i]$. For the workload as a whole, we calculated an array $\overline{CSS}$ to average out the CSS-arrays of all the instances; that is, for each $i \geq 1$, $\overline{CSS}[i]$ was the average $CSS[i]$ of all the instances in the workload.

Figure 6b plots the $\overline{CSS}$ array for `ordered-interleave` and `top-down`. Note that the y-axis is in log scale. It is evident that `ordered-interleave` was significantly faster in reducing CSS. In particular, the average CSS was below 10 in less than 40 questions, while at this cost `top-down` still had an average CSS close to 10,000.

The next experiment aims to provide a "zoom-in" into the cost of an algorithm on individual instances in a workload. Towards the purpose, let us define the *sum of out-degrees of ancestors* (SODA) of a node u as the total out-degree of all the proper ancestors of u. For `Amazon`, the SODA values of all the leaves fell in the range $[84, 399]$. We cut the range into 20 intervals of the same length. For each algorithm, we measured 20 costs, one for each interval I. Specifically, the measurement on I was the algorithm's average cost on all the instances that were defined by the leaves with SODA values in I. By putting these 20 averages together, we acquired a cost distribution of the algorithm over the SODA spectrum.

Figure 6c compares the obtained cost distributions of `ordered-interleave` and of `top-down`. The former algorithm consistently outperformed the latter in all intervals, often by large factors. Note that there were no results for the SODA range from 212 to 308, because no leaves have SODA values in that range. Also, observe that the cost of `top-down` exhibited a clear ascending trend as the SODA value increased.

We repeated the same experiment on `ImageNet`, using `DFS-interleave` and `top-down` as the competitors. Figure 7 presents the results, which were obtained in the same manner as those of Figure 6. A bit extra explanation is needed regarding the SODA of a node u in a DAG. Let us define a *proper ancestor* of u as a node v that has a path reaching u. With this notion, SODA becomes well defined also for a DAG,

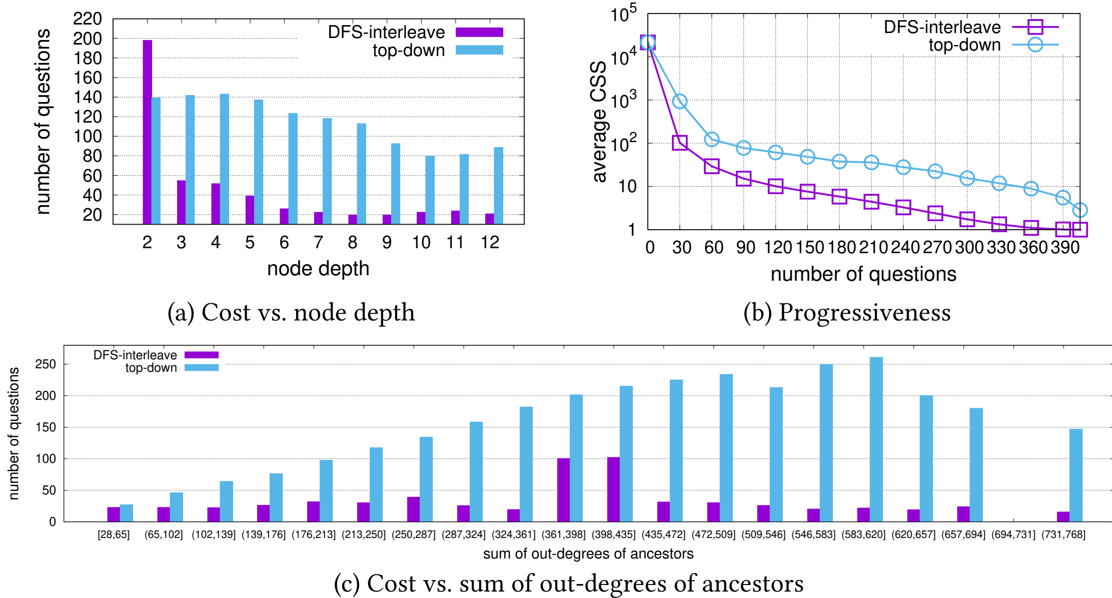


(a) Cost vs. node depth

(b) Progressiveness

(c) Cost vs. sum of out-degrees of ancestors

Figure 6: IGS on `Amazon` (i.e., $k = 1$)



(a) Cost vs. node depth

(b) Progressiveness

(c) Cost vs. sum of out-degrees of ancestors

Figure 7: IGS on `ImageNet` (i.e., $k = 1$)

thus allowing us to generate Figure 7c in the way Figure 6c was produced.

Two comments are worth noting. First, in Figure 7a, there were no results at depth 1, because `ImageNet` has no leaves at this depth. Second, the cost of `top-down` initially increased with SODA, but the trend of increasing disappeared after SODA had become large enough. This can be explained by the fact that, in a DAG, there can be multiple paths from the root to the target leaf; and a large SODA can be caused by an abundance of such paths, which actually makes it more likely for `top-down` to find a relatively short way to get to the target. Other than the above, the overall observations are similar to those on `Amazon`. Notice that the performance advantages of our solution were even more prominent on `ImageNet`.

A final remark concerns the CPU efficiency. The most computation-intensive step is the preparation of the heavy-path DFS-tree in Section 5.1. But even this step took no more

than 10 seconds on both datasets. The CPU delays in the other steps were all unnoticeable. The same was true in all the other experiments to be reported in this paper.

Results on k-IGS. The experiment results on k-IGS were in general similar to those of the experiments presented earlier (for $k = 1$), and can be found in Appendix F.

7.2 IGS on Crowdsourcing

The results in Section 7.1 are representative of what one would expect in applications (such as those in Section 2.3) where the oracle is reliable. In this subsection, we will inspect the usefulness of `DFS-interleave` in crowdsourcing scenarios, where questions are answered by human workers that could err, thus generating "noise" that may prevent an algorithm from returning a correct answer. As our algorithmic framework in Section 2.1 does not explicitly take mistakes into account, `DFS-interleave` is not tailored for crowdsourcing. Nevertheless, the subsequent evaluation aims to make three points. First, `DFS-interleave` was resilient to *random noise* (i.e., mistakes due to carelessness) such that it achieved good accuracy even in its current form. Second, the main difficulty in crowdsourcing seemed to stem from the *system noise* caused by humans' lack of knowledge about the objects of concern. Third, `top-down` was much more susceptible to noise simply because it needed to issue more questions.

Data. We again took the Amazon dataset, but in its DAG form. Recall the tree hierarchy generated in Section 7.1, i.e., the product category tree at Amazon. We observed that, to some products p, the source file *metadata.json.gz* attached multiple categories *cat*, each corresponding to a leaf node in the category tree. By inserting p as a new leaf and adding an edge from node *cat* to p, we converted the tree hierarchy into a DAG.

Two issues, however, arose. First, some products had missing values in the "title" field, and therefore, could not be posted as informative queries on a crowdsourcing platform (this will be further clarified later). Second, the file contained over 9 million products, such that some nodes in the DAG ended up with unrealistically huge out-degrees. We remedied the issues as follows. We cleansed the dataset by discarding the products with empty title fields. This still left over five million products such that the second issue still existed. We sorted those products by id, and then picked 25,000 products *evenly* from the sorted list (i.e., picking the $i \cdot \lfloor x/25000 \rfloor$-th for each $i \in [1, 25000]$, where x was the number of products after cleansing). Then, we created a DAG in the way described earlier, by ranging p over the 25,000 products, and removed "dead" category nodes with no product descendants. This yielded a DAG hierarchy with 33,573 nodes in total.

Crowd. The DAG thus generated allowed us to evaluate the two competing IGS algorithms (i.e., `DFS-interleave` and `top-down`) on the commercial crowdsourcing platform of *figure-eight.com*, in a scenario where one would like to leverage the crowd to automatically assign pictures to products. Specifically, given a product z (e.g., a US history book), we defined its *metainfo* as the combination of (i) a picture[2] of z, (ii) the title of z, and (iii) the detailed description of z (if such description existed in the source file). In a *question* posted on *figure-eight.com*, we provided the metainfo of z, and asked: "*does this product belong to the following category?*" Each category in the question is a node in the DAG (e.g., [*, Books, Comics], in which case the correct answer for "z = a history book" should be *no*). Interestingly, z itself offered the *ground truth* such that we could directly compare z to the output of the algorithm to see if it was *correct*.

Two standard measures were taken to ensure good quality for the answers collected. First, every human worker had to pass a so-called *gold standard test* where s/he was given a list of questions and must correctly resolve 85% to be qualified. Second, for the same question, confidence-guided repeats and majority-taking were applied: more answers were solicited until either a maximum of 9 answers had been returned or a confidence at least 0.7 had been reached from the majority of at least 5 answers[3].

Workload. As in Section 7.1, each leaf (a.k.a. product) in the DAG defines an *instance* of IGS. We generated a *workload* of 50 instances scattered evenly in the spectrum of SODA values defined in Section 7.1. Specifically, the leaves of the DAG had SODA values in the range $[40, 2255]$. We partitioned the range into 10 equi-length intervals. For each interval, we sorted the the leaves covered in that interval by SODA value, and picked 5 leaves evenly from the sorted list. This gave 50 leaves, a.k.a. instances, in total, which constituted the workload.

Metrics. For each instance in the workload, we compared `top-down` and `DFS-interleave` using two metrics: (i) whether the algorithm correctly solved the instance, and (ii) if so, what was the *crowd cost*, defined as the total number of answers collected throughout the algorithm (if the same question received x answers, the crowd cost increased by x). For fairness, both algorithms were executed on the same DAG, where the out-neighbors of each node were ordered randomly.

Results. Table 2 details the performance of `top-down` and `DFS-interleave` on each of the 50 instances in the workload. Recall that five instances were created from each of the 10 intervals that partition the SODA spectrum. Those 10

[2] The source file included a picture URL for each of the 25,000 products.
[3] Such a functionality was directly available at *figure-eight.com*.

id	SODA range	top-down			DFS-interleave		
		crowd cost	success rate	avg crowd cost on successful instances	crowd cost	success rate	avg crowd cost on successful instances
1		–			–		
2		–			–		
3	[40, 261]	–	40%	707	188	40%	131
4		390			–		
5		1023			74		
6		509			90		
7		1061			185		
8	[262, 483]	–	60%	991	–	80%	410
9		1403			1276		
10		–			90		
11		–			457		
12		–			885		
13	[484, 705]	–	0%	–	480	80%	477
14		–			85		
15		–			–		
16		–			830		
17		–			69		
18	[706, 927]	–	20%	674	–	60%	374
19		–			225		
20		674			–		
21		–			4248		
22		–			1484		
23	[928, 1149]	–	0%	–	4606	100%	2410
24		–			1350		
25		–			362		
26		–			346		
27		–			545		
28	[1150, 1371]	–	0%	–	806	100%	762
29		–			1769		
30		–			344		
31		729			–		
32		514			165		
33	[1372, 1593]	–	80%	701	–	60%	161
34		1021			128		
35		537			189		
36		1296			409		
37		–			520		
38	[1594, 1815]	–	40%	864	680	80%	425
39		432			90		
40		–			–		
41		1122			187		
42		1712			790		
43	[1816, 2037]	–	60%	1527	–	80%	520
44		–			258		
45		1748			845		
46		2002			1095		
47		–			1355		
48	[2038, 2259]	–	60%	1649	745	80%	915
49		1566			–		
50		1378			465		

Table 2: Results of the workload on crowdsourcing

intervals are listed in the second column of the table. The instances from the same interval form a *group*. Columns 3-5 concern only `top-down`. Specifically, for each instance, Column 3 gives the crowd cost of `top-down`, but only if the algorithm managed to resolve the instance; an incorrect output of the algorithm is indicated by the sign "–". For each group, (i) the percentage in Column 4 is calculated as $x/5$, where x is the number of instances in the group that were correctly resolved by `top-down`, while (ii) the number in Column 5 is the average crowd cost of the algorithm on those x instances. Columns 6-8 depict `DFS-interleave` in the same manner.

`DFS-interleave` successfully resolved 38 instances, striking an overall success rate of 76%. In contrast, `top-down` managed with only 18 instances, settling with an overall success rate of only 36%. To explain such a vast difference, first note that since every question has a chance of triggering a fatal mistake, the overall failure probability increases with the number of questions. The gain in accuracy achieved by `DFS-interleave`, therefore, can be attributed to the fact that it necessitated much fewer questions than `top-down`, as can be clearly seen from the table. It is worth pointing out that the crowd-cost comparison between the two algorithms is reminiscent of the patterns observed earlier in Section 7.1.

We delved into each of the 12 instances that `DFS-interleave` failed to resolve. It turned out that *none* of those cases were due to "careless" mistakes by the human workers. Indeed, even though such kind of mistakes *did* happen, their influence was essentially eliminated by the quality control measures adopted. In other words, random noise hardly played any roles in the outcome of the algorithm. This, at least in retrospect, was not surprising, and essentially confirmed the effectiveness of quality control at a modern crowdsourcing site such as *figure-eight.com.*

So, what *was* the cause behind the mistakes made by `DFS-interleave`? Next, we gave three most representative causes. Interestingly, none of these causes was really the fault of the human workers. Phrased differently, those causes correspond to system noise that is difficult to deal with, and therefore would be persistent regardless of the IGS algorithm applied.

Cause 1: incomplete ground truth. One mistaken instance of `DFS-interleave` is on a product with the title "Crocs Women's Nadia Boot".[4] The product is a type of footwear that extends almost to laps. The ground truth places it under the category [*, Clothing, Shoes & Jewelry, Women, Shoes, Fashion Sneakers]. However, all the workers classified the product into the category [*, Clothing, Shoes & Jewelry, Women, Shoes, Boots], which also appears reasonable, and could have been added to the ground truth.

Cause 2: questionable ground truth. Another instance mistaken by `DFS-interleave` is on a product with the title "Naturalizer Women's Lennox Pump".[5] The product is a pair of high-heels suitable even for business meetings. The ground truth, however, dictates that it should be under the category [*, Clothing, Shoes & Jewelry', Women, Shoes, Sandals], which no workers were able to discern.

Cause 3: subjective judgments. Our last example is on a product titled "Kenneth Cole New York 'Modern Ombre' Blue Green Ombre Resin Linear Earrings".[6] By the ground truth, it is under the category [*, Clothing, Shoes & Jewelry, Women, Jewelry, Fashion]. In contrast, many workers chose [*, Clothing, Shoes & Jewelry, Women, Jewelry, Fine]. Note that the subtle difference is about whether the jewelry piece is "fine" or "fashion". This appears to be a rather subjective as one can see from the image at the URL provided earlier in the footnote.

8 RELATED WORK

Most relevant to our paper is the work of [15], which as mentioned in Section 1 introduced the offline counterpart of IGS (under the name *human-assisted graph search*). The solutions in [15] were designed for the scenario where the algorithm must ask all the questions altogether. The number of questions generated by those solutions is huge: often at the same magnitude as the number of nodes in the input hierarchy. As explained in Section 1, the main advantage of IGS (owing to the possibility of interaction) is that the number of questions can be reduced dramatically.

At a higher level, our work is somewhat related to *human-based computation* (HBC). The fundamental rationale behind this area is that *some tasks are inherently easy for humans*, as opposed to the old computing philosophy that "computation is a job of machines". HBC algorithms aim at engaging both humans and machines so that they can work collaboratively to solve a problem effectively and/or efficiently. In recent years, considerable attention has been devoted to *crowdsourcing*, which is a *large-scaled* form of HBC that involves a huge number of human workers. A significant amount of work has been carried out on studying crowdsourcing algorithms (see representative works [2–4, 6, 8, 10, 11, 14, 19]) and on developing crowdsourcing systems (see representative works [5, 9, 12, 16]).

Two remarks are in order about interpreting our work as a form of HBC. First, there is not much "computation" by the traditional yardstick of HBC: all an IGS algorithm does is to figure out the node that a human has in mind. The challenge lies in how to utilize *reachability* to identify that node as quickly as possible. Second, our algorithms are designed for an authoritative oracle that never errs. This implies opportunities for improving those algorithms in terms of effectiveness on a crowdsourcing platform. In fact, some crowdsourcing-specific issues have been experimentally identified in Section 7.2. Integrating our algorithms with remedies to those issues would make a promising direction for future work.

9 CONCLUSIONS

Conventionally, people are used to searching a decision tree/graph in the straightforward top-down fashion. This paper aims to show that there can be alternative strategies achieving better efficiency than that traditional wisdom. To allow for a rigorous algorithmic study, we introduced the the *interactive graph search problem. Here, the input is a directed acyclic graph G. Given an initially unknown vertex z in G, the objective is to eventually locate z by asking* reachability questions*: each question* specifies a query node q and obtains a boolean answer as to *whether* z is reachable from q. We have described algorithms which solve variants of the problem using a provably small number of questions, and established a nearly matching lower bound. We have also presented an experimental evaluation to demonstrate the efficiency and usefulness of the proposed solutions in real world scenarios.

[4] Image at *ecx.images-amazon.com/images/I/41z0zj%2BVhzL._SY395_.jpg.*
[5] Image at *ecx.images-amazon.com/images/I/41lVbFn%2B2lL._SX395_.jpg*
[6] Image at *ecx.images-amazon.com/images/I/31S4Sgi-HoL._SY300_.jpg.*

ACKNOWLEDGEMENTS

The research of Yufei Tao was partially supported by a direct grant (Project Number: 4055079) from CUHK and by a Faculty Research Award from Google. The research of Guolinag Li was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61521002, 61661166012), Huawei, and TAL education.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition.* The MIT Press.

[2] Susan B. Davidson, Sanjeev Khanna, Tova Milo, and Sudeepa Roy. 2013. Using the crowd for top-k and group-by queries.. In *ICDT*. 225–236.

[3] Eyal Dushkin and Tova Milo. 2018. Top-k Sorting Under Partial Order Information. In *SIGMOD*. 1007–1019.

[4] Ju Fan, Guoliang Li, Beng Chin Ooi, Kian-Lee Tan, and Jianhua Feng. 2015. iCrowd: An Adaptive Crowdsourcing Framework. In *SIGMOD*. 1015–1030.

[5] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*. 61–72.

[6] Stephen Guo, Aditya G. Parameswaran, and Hector Garcia-Molina. 2012. So who won?: dynamic max discovery with the crowd. In *SIGMOD*. 385–396.

[7] Ruining He and Julian McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *WWW*. 507–517.

[8] Chien-Ju Ho, Shahin Jabbari, and Jennifer Wortman Vaughan. 2013. Adaptive Task Assignment for Crowdsourced Classification. In *ICML*. 534–542.

[9] Guoliang Li, Chengliang Chai, Ju Fan, Xueping Weng, Jian Li, Yudian Zheng, Yuanbing Li, Xiang Yu, Xiaohang Zhang, and Haitao Yuan. 2017. CDB: Optimizing Queries with Crowd-Based Selections and Joins. In *SIGMOD*. 1463–1478.

[10] Xuan Liu, Meiyu Lu, Beng Chin Ooi, Yanyan Shen, Sai Wu, and Meihui Zhang. 2012. CDAS: A Crowdsourcing Data Analytics System. *PVLDB* 5, 10 (2012), 1040–1051.

[11] Adam Marcus, David R. Karger, Samuel Madden, Rob Miller, and Sewoong Oh. 2012. Counting with the Crowd. *PVLDB* 6, 2 (2012), 109–120.

[12] Adam Marcus, Eugene Wu, Samuel Madden, and Robert C. Miller. 2011. Crowdsourced Databases: Query Processing with People. In *CIDR*. 211–214.

[13] Jonathan J Oliver. 1993. Decision Graphs - An Extension of Decision Trees. In *Int. Conf. Artificial Intelligence and Statistics*. 343–350.

[14] Aditya G. Parameswaran, Hector Garcia-Molina, Hyunjung Park, Neoklis Polyzotis, Aditya Ramesh, and Jennifer Widom. 2012. CrowdScreen: algorithms for filtering data with humans. In *SIGMOD*. 361–372.

[15] Aditya G. Parameswaran, Anish Das Sarma, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. 2011. Human-assisted graph search: it's okay to ask questions. *PVLDB* 4, 5 (2011), 267–278.

[16] Hyunjung Park, Richard Pang, Aditya G. Parameswaran, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. 2012. Deco: A System for Declarative Crowdsourcing. *PVLDB* 5, 12 (2012), 1990–1993.

[17] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh K. Mohania. 2008. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM*. 13–22.

[18] Daniel Dominic Sleator and Robert Endre Tarjan. 1983. A Data Structure for Dynamic Trees. *JCSS* 26, 3 (1983), 362–391.

[19] Vasilis Verroios, Hector Garcia-Molina, and Yannis Papakonstantinou. 2017. Waldo: An Adaptive Human Interface for Crowd Entity Resolution. In *SIGMOD*. 1133–1148.

[20] Ka-Ping Yee, Kirsten Swearingen, Kevin Li, and Marti A. Hearst. 2003. Faceted metadata for image search and browsing. In *CHI*. 401–408.

A PROOF OF LEMMA 2

Let x be the number of iterations performed by `ordered-interleave`. For each $i \in [1, x]$, we denote by $d_i \le d - 1$ the number of questions issued at Line 4 in the i-th iteration. Equivalently, d_i is the number of child nodes of u that are queried at Line 4.

As in `interleave`, the binary search at Line 3 asks at most $\lceil \log_2 h \rceil$ questions for each iteration. It thus follows that `ordered-interleave` entails a cost at most

$$\lceil \log_2 h \rceil \cdot x + \sum_{i=1}^{x} d_i.$$

Since Π has at most $1 + \lfloor \log_2 n \rfloor$ levels (Lemma 1), obviously $x \le 1 + \lfloor \log_2 n \rfloor$. It suffices to prove an upper bound for $\sum_{i=1}^{x} d_i$.

For each $i \in [1, x]$, let v_i be the "child v" identified at Line 4 in the i-th iteration. Denote by n_i the subtree size of v_i. Specially, define v_0 as the root r of T, and $n_0 = n$.

Lemma 6. $n_i \le n_{i-1}/(d_i + 1)$.

Proof. In executing the i-th iteration, Line 3 binary searches a path π to identify a node u in π. Since v_{i-1} is the first node on π, we know that u must be in the subtree of v_{i-1}. Hence, the subtree size of u is at most n_{i-1}.

At Line 4 (applying the modification in Section 4.2 that searches the child nodes of u in non-ascending order of subtree size), since d_i child nodes of u were queried until v_i is found, u has at least d_i child nodes whose subtrees are as large as that of v_i (counting also the child node of u in π). The lemma then follows. □

Lemma 7. $(d_1 + 1)(d_2 + 1)...(d_x + 1) \le n$.

Proof. By Lemma 6, we know that

$$n_x \quad \le \quad \frac{n}{(d_1 + 1)(d_2 + 1)...(d_x + 1)}.$$

The lemma then follows from $n_x \ge 1$. □

Now it remains to upper bound $\sum_{i=1}^{x} d_i$ subject to the above condition. The lemma below provides such an upper bound, which will complete the proof.

Lemma 8. $\sum_{i=1}^{x} d_i \le (d - 1) \cdot \lceil \log_d n \rceil$.

Proof. We prove that the lemma holds even if $d_1, ..., d_x$ are non-negative real values. Without loss of generality, assume $d_1 \ge d_2 \ge ... \ge d_x$.

Claim: Fix the value of x. To maximize $\sum_{i=1}^{x} d_i$ subject to Lemma 7, the best strategy is to set

- variables $d_1, \ldots, d_i$ to $d-1$ for some i;
- optionally d_{i+1} to a value greater than 0 but less than $d-1$;
- and the remaining variables all to 0.

Proof of the claim: Suppose that $\sum_{i=1}^{x} d_i$ is maximized when d_{i-1} and d_i are both greater than 0 but less than $d-1$, for some $i \geq 2$. Let $s = (d_{i-1}+1)(d_i+1)$. Clearly, $1 < s < d^2$.

- If $d < s < d^2$, we set d_{i-1} to $d-1$, and d_i to $\frac{s}{d} - 1$. The new values still satisfy Lemma 7, but increase $\sum_{i=1}^{x} d_i$, contradicting the fact that $\sum_{i=1}^{x} d_i$ is already maximized.
- If $s \leq d$, we set d_{i-1} to $s-1$, and d_i to 0. The new values still satisfy Lemma 7, but increase $\sum_{i=1}^{x} d_i$, i.e., contradiction. **QED**

Now we vary x. When $x \leq \lceil \log_d n \rceil$, by the above claim $\sum_{i=1}^{x} d_i \leq (d-1)x \leq (d-1) \cdot \lceil \log_d n \rceil$.

When $x \geq \lceil \log_d n \rceil + 1$, by the above claim $\sum_{i=1}^{x} d_i$ is maximized by setting

- d_i to $d-1$ for $1 \leq i \leq \lfloor \log_d n \rfloor$;
- (only if n is not a power of d) $d_{1+\lfloor \log_d n \rfloor}$ to a value larger than 0 and less than $d-1$;
- and the remaining variables to 0.

It thus follows that $\sum_{i=1}^{x} d_i \leq (d-1)x \leq (d-1) \cdot \lceil \log_d n \rceil$. □

B PROOF OF LEMMA 3

The hard hierarchy is simply a perfect d-ary tree T with $h+1$ levels, where $h = \lfloor \log_d n \rfloor$.

We let the adversary — Bob — play the role of oracle. He does not choose the target node z at the beginning. Instead, he observes how to algorithm runs, and gradually shrinks the set of nodes where he could place z, without violating any of the answers he (as the oracle) has given to the algorithm's questions so far. He will execute a strategy of h rounds, where in each round he forces the algorithm to ask at least $d-1$ questions.

Bob's strategy adheres to the following invariant: at the beginning of a round, he has chosen a node u, and made up his mind to place z in the subtree of u eventually (for round 1, u is simply the root of T). At the end of the round, he will descend into a child node of u, and set u to that child node for the next round.

Now we explain the details of Bob's actions in a round. Let S be the set of d child nodes of u. Suppose that Alice asks a question with query node q. Bob answers the question as follows:

- If q is not in the subtree of u, he returns $reach(q) = no$.
- If $q = u$, he returns $reach(q) = yes$.
- If q is in the subtree of a child node v of u, he returns $reach(q) = no$. Furthermore, if v is still in S, he removes v from S.

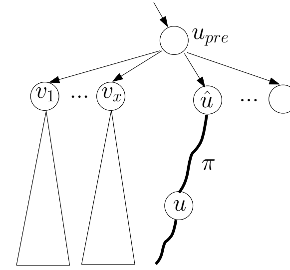


Figure 8: Proof of Lemma 9

The round finishes when $|S|$ has decreased to 1. In this case, he sets u to the only node left in S. With this u, the next round starts.

After h rounds, u is a leaf node in T. Bob then chooses u as the target node z.

In each round, Alice needs to ask at least $d-1$ questions in order to shrink $|S|$ from d to 1. Therefore, in the whole process, Alice must ask at least $(d-1) \cdot h = (d-1) \cdot \lfloor \log_d n \rfloor$ questions.

C PROOF OF LEMMA 4

Consider the moment when v_1 was discovered. By our ordering strategy, $count(v_1) \geq count(v_2)$ at that moment. By the white-path theorem (Theorem 1), we know that the subtree of v_1 of T has a size equal to precisely $count(v_1)$.

What is the subtree size of v_2? By the white-path theorem, it is exactly the value of $count(v_2)$ at the moment when v_2 was discovered, which is *after* the discovery of v_1. As $count(v_2)$ cannot increase during the algorithm, we conclude that the subtree size of v_1 is at least that of v_2.

D CORRECTNESS OF DFS-INTERLEAVE

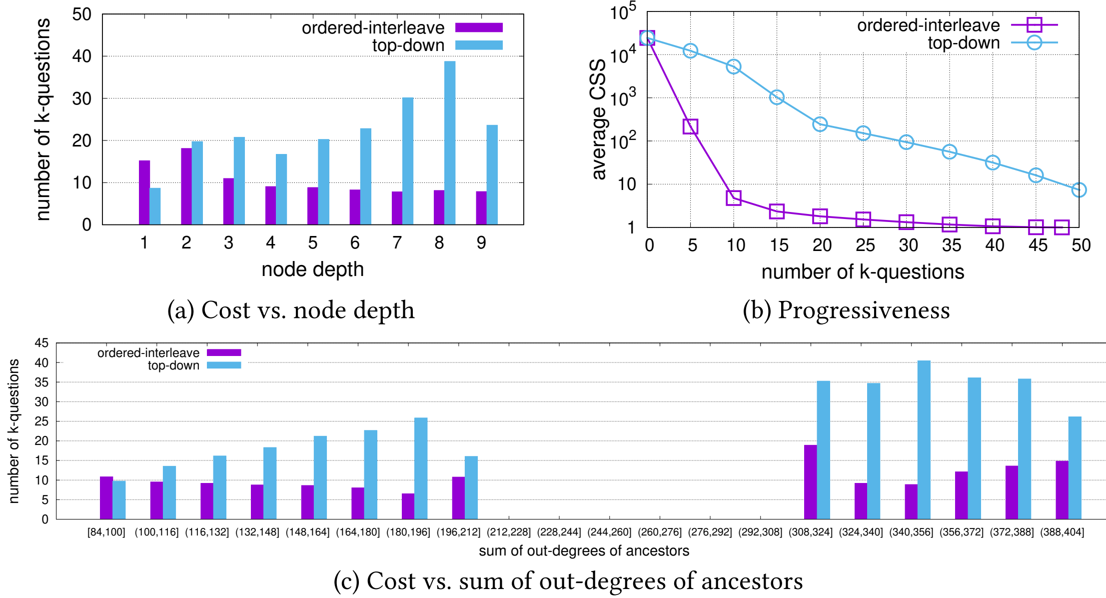
As before, let G be the input DAG hierarchy, V the set of vertices in G, and T the heavy-path DFS-tree decided in Section 5.1. Given a node $u \in V$, define $P(u)$ as the set of nodes $u' \in V$ satisfying:

- u' was discovered earlier than u in the DFS described in Section 5.1, and
- u' is not an ancestor of u in T.

Example. Consider again the example shown in Figure 5. P(node 7), for instance, consists of nodes 8, 10, 11, 6. As another example, P(node 13) = {4, 8, 10, 11, 6, 7, 12, 14}. □

LEMMA 9. *Consider any node u obtained at Line 4 of* `DFS-interleave`. *None of the nodes in $P(u)$ can reach the target node z.*

PROOF. Call node u a *pivot node*. Also, let us refer to Lines 3-7 as an *iteration*. We will prove the lemma by induction on the number of iterations.

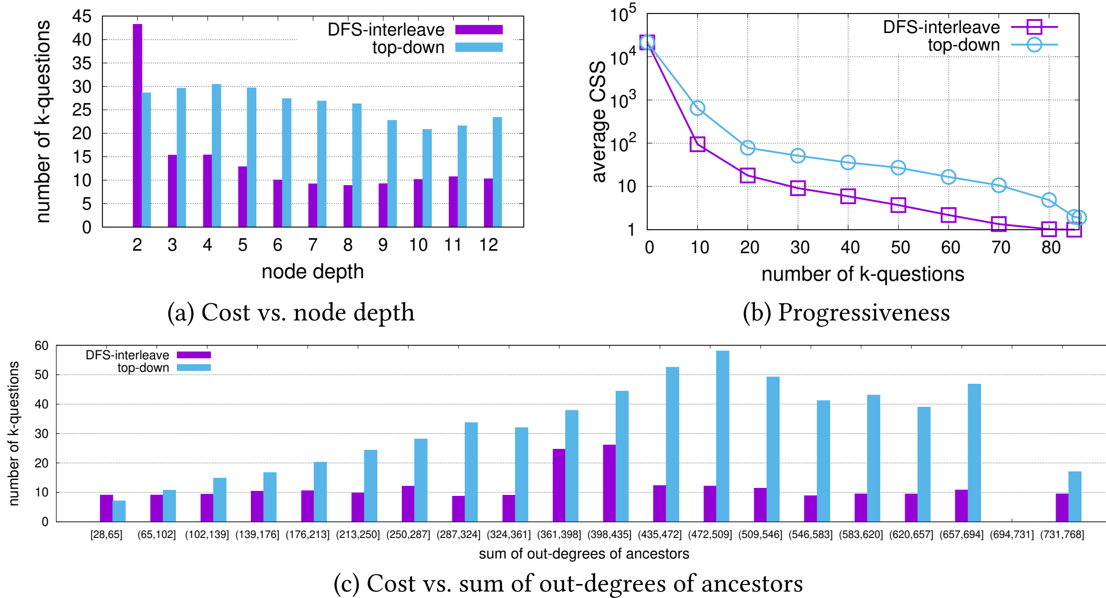


(a) Cost vs. node depth

(b) Progressiveness

(c) Cost vs. sum of out-degrees of ancestors

Figure 9: k-IGS on `Amazon` with $k = 5$



(a) Cost vs. node depth

(b) Progressiveness

(c) Cost vs. sum of out-degrees of ancestors

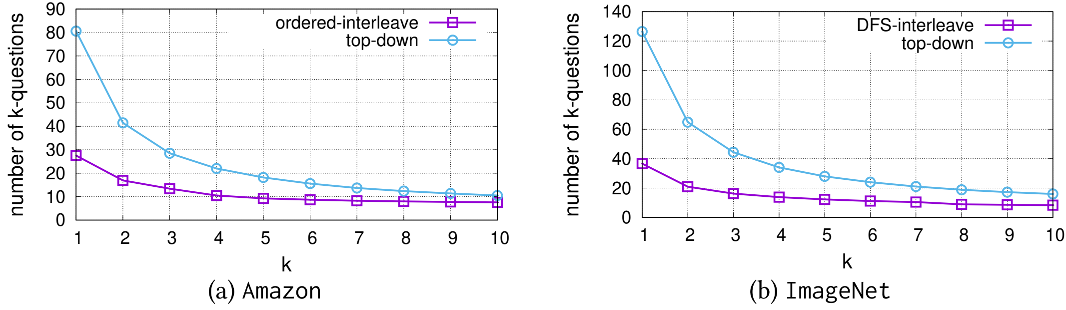
Figure 10: k-IGS on `ImageNet` with $k = 5$

First Iteration. In this case $P(u) = \emptyset$ noticing that π is the leftmost root-to-leaf path of T. The claim obviously holds.

Inductive Step: Iteration $i \geq 2$. Let u_{pre} be the pivot node obtained in Iteration $i-1$. The node $\hat{u}$ at Line 3 of the i-th iteration (which is also the node v at Line 5 of Iteration $i-1$) is a child node of u_{pre}. The pivot node of this iteration is on the leftmost $\hat{u}$-to-leaf path π of T. See Figure 8 for an illustration, where $v_1, \ldots, v_x$ ($x \geq 0$) are the child nodes of u_{pre} to the left of $\hat{u}$.

What are the nodes in $P(u) \setminus P(u_{pre})$? Remember that a pre-order traversal of T enumerates the nodes exactly in the order they were discovered in DFS. Therefore, $P(u) \setminus P(u_{pre})$ is exactly the set of nodes that are in the subtrees of $v_1, \ldots, v_x$ as shown in the figure.

By the way our algorithm runs, we know that $\hat{u}$ is the leftmost child v of u_{pre} with $reach(v) = yes$. It thus follows that none of the nodes $v_1, \ldots, v_x$ can reach the target node z; and therefore, neither can any of their descendants.



(a) Amazon (b) ImageNet

Figure 11: k-IGS: Cost vs. k

By the inductive assumption, none of the nodes in $P(u_{pre})$ can reach z. We thus conclude that no nodes in $P(u)$ can reach z, completing the proof. □

We now return to the correctness of DFS-interleave. It suffices to show that when Line 4 finds no child v of u satisfying $reach(v) = yes$, u must be the target node (and hence, is correctly returned at Line 5).

Assume that this is not true, i.e., u is not the target node z. Then, since $reach(u) = yes$, the out-neighbor property (Proposition 1) tells us that u must have an out-neighbor u' that can reach z. Consider the edge (u, u'). As discussed in Section 3.2, every edge in G can be classified as a tree edge, a forward edge, or a cross edge. We know that (u, u') is not a tree edge; otherwise, the algorithm would have found u' as a child of u in T. It cannot be a forward edge either, because in that case, still the algorithm would have found a child v of u with $reach(v) = yes$. Hence, (u, u') must be a cross edge.

Since u' is not a descendant of u in T, the white-path theorem (Theorem 1) tells us that u' was discovered before u. Furthermore, u' cannot be an ancestor of u in T (there can be no cycles). Therefore, $u' \in P(u)$. However, Lemma 9 asserts that the target node z cannot be reachable from u', giving a contradiction.

E PROOF OF LEMMA 5

Similar to Appendix A, define

- x as the number of iterations performed by ordered-interleave;
- for each $i \in [1, x]$, $d_i \leq d - 1$ as the number of child nodes of u that are queried at Line 4 in the i-th iteration.

It thus follows that ordered-interleave entails a cost at most

$$
\begin{aligned}
& \lceil \log_k h \rceil \cdot x + \sum_{i=1}^{x} \lceil d_i / k \rceil \\
\leq \quad & \lceil \log_k h \rceil \cdot x + \sum_{i=1}^{x} (1 + d_i / k) \\
\leq \quad & (1 + \lceil \log_k h \rceil) \cdot x + \frac{1}{k} \sum_{i=1}^{x} d_i
\end{aligned}
$$

By $x \leq 1 + \lfloor \log_2 n \rfloor$ (Lemma 1) and Lemma 8, we complete the proof.

F EXPERIMENTS ON k-IGS

We now proceed to the k-IGS problem in Section 6. Remember that an oracle in this problem is more powerful, in the sense that each time it can reveal the reachability of k nodes to the target node.

Repeating the experiments of Figures 6 and 7 but setting $k = 5$, we obtained the results in Figures 9 and 10 for Amazon and ImageNet, respectively. The behavior of all algorithms and their relative superiority were very similar to what was observed in Section 7.1.

The last experiment inspected the influence of k on the algorithms' cost. Focusing on ImageNet, Figure 11a plots the average (per-instance) cost of ordered-interleave in processing a workload as k grew from 1 to 10, and also the same for top-down. Turning to ImageNet, Figure 11b presents the corresponding results with respect to DFS-interleave and top-down. As expected, all algorithms had their costs improved continuously as k got higher, confirming our theoretical analysis.