Interactive Graph Search Made Simple

SHANGQI LU, Hong Kong University of Science and Technology (Guangzhou), China
RU WANG, The Chinese University of Hong Kong, China
YUFEI TAO, The Chinese University of Hong Kong, China

Interactive graph search (IGS) has proven to be a useful information retrieval paradigm in a diverse set of applications. Robust IGS algorithms are notoriously difficult to design because they are deeply rooted in graph theory. The current state-of-the-art algorithms either fail to achieve an optimal number of interaction rounds or rely on interfaces demanding tedious user inputs. Furthermore, previous research has paid little attention to the underlying computation bottleneck, which is currently dealt with using primitive implementations. This work remedies the above issues altogether. Utilizing novel findings on the problem characteristics, we develop an algorithmic framework for IGS that requires a designer to fill in the details for only two "black-box" operations. Our framework, when instantiated with surprisingly simple black-box implementations, yields optimal algorithms not only in all the scenarios explored before but also in new scenarios never studied. We accompany our framework, designed to minimize interaction rounds, with a new algorithm designed to reduce the CPU time complexity significantly. Extensive experiments on both real and synthetic data confirm both the efficacy and efficiency of the proposed techniques.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; • **Information systems → Collaborative search**.

Additional Key Words and Phrases: Interactive Graph Search; Graph Algorithms

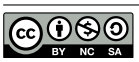
ACM Reference Format:
Shangqi Lu, Ru Wang, and Yufei Tao. 2025. Interactive Graph Search Made Simple. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 177 (June 2025), 25 pages. https://doi.org/10.1145/3725409

1 Introduction

Interactive graph search (IGS), introduced in [22], is an iterative questioning procedure between an algorithm and an adversary, known as the *oracle*. The procedure operates on a directed acyclic graph (DAG) $G = (V, E)$ with a single root (i.e., a vertex with no incoming edges). At the outset, the oracle selects an arbitrary vertex $t \in V$ to serve as the *target*. The algorithm's task is to identify this target by issuing *queries* to the oracle repeatedly. In each query, the algorithm chooses a vertex u and inquires: *can u reach t*, or equivalently, *is there a path from u to t within G*? The oracle responds with either yes or no. The algorithm can continue querying until the target is found. Its *cost* is defined as the total number of queries made.

The procedure of IGS models practical situations where the goal is to uncover a hidden target vertex t in a DAG, with the main action available being to pick a vertex u from the graph and examine it against a *reachability-consistent predicate* — meaning a condition that holds true if and only if u can reach t. This search paradigm emerges in diverse scenarios including image categorization [19], causal analysis in machine learning [10], distributed file systems [18], software

Authors' Contact Information: Shangqi Lu, shangqilu@hkust-gz.edu.cn, Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China; Ru Wang, rwang21@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Yufei Tao, taoyf@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China.



ACM 2836-6573/2025/6-ART177
https://doi.org/10.1145/3725409

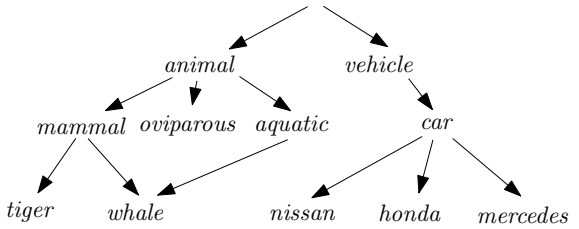


Fig. 1. IGS for image categorization (adapted from [19, 22])

testing [2], and even the design of new data structures [16]. These multifaceted applications have generated significant research interest in IGS in recent years [3, 4, 14–16, 22, 25, 26].

For a concrete discussion, let us revisit a well-known image categorization scenario from [19]. Machine learning research often requires images annotated by humans using labels chosen from a sophisticated hierarchy of concepts. Figure 1 shows a portion of such a hierarchy where the label of a node u is a generalization of the label of any node that can be reached from u. The objective is to assign the most specific label to a given image. For this purpose, an algorithm selects a node u with label x and asks a human: *is the image an* x? The human acts as the oracle, and interestingly, even though the human did not choose the target, s/he can still provide a correct answer with ease. For example, consider an image of a tiger. A clever algorithm would pick the nodes `vehicle`, `mammal`, and `tiger` (in this order) and, after receiving the human's answers "no", "yes", and "yes", can correctly identify the target `tiger` as the most informative label.

The number of queries, however, can be rather large, e.g., the experiments of [16] showed that the number can reach several hundred on real data. This poses a serious issue because a high cost demands a lengthy sequence of interaction rounds that could challenge the composure of even the most patient human annotator. A common approach to address the issue is to allow each query to solicit reachability information for multiple vertices. Specifically, each query now presents a set Q of k vertices from V. For each vertex $u \in Q$, the oracle reveals whether u can reach the target t. In Figure 1, for example, a query of $k = 3$ would specify $Q = \{$`vehicle`, `mammal`, `tiger`$\}$. Upon acquiring human answers {no, yes, yes}, an algorithm can pinpoint the target `tiger` with cost 1. This was exactly the version of IGS initially proposed by Tao et al. [22]. Today, the problem has been resolved optimally. Lu et al. [15, 16] developed an algorithm that guarantees locating the target in $O(\log_k n + (d/k)\log_d n)$ queries, where $n = |V|$ and d is the maximum vertex out-degree in G. Furthermore, they also proved that this cost is asymptotically optimal in the worst case.

Nevertheless, the above oracle — called the **classical oracle** in the literature — also suffers from a drawback: it is not user-friendly because it requires a human to provide too many answers. When k increases, answering a query soon becomes burdensome because the human would need to click on many (up to k) buttons. This is problematic, considering that effectively reducing the number of interaction rounds calls for large values of k. Motivated by this, Lu et al. [16] introduced the notion of *one-click* oracles, which require the human to click on only one button per query. They formulated such an oracle — aptly named the **taciturn oracle** — which (like the classical oracle) still accepts a set Q of k vertices, but (unlike the classical oracle) provides only a binary answer: whether Q has at least one vertex capable of reaching the target. For example, in Figure 1, given a query $Q = \{$`vehicle`, `mammal`, `tiger`$\}$, the taciturn oracle would just respond "yes", without indicating which vertices in Q can reach the target. For this new oracle, Lu et al. [16] described an algorithm that finds the target with $O(\log n \cdot \log k + (d/k)\log_d n)$ queries. Note that this is more expensive than how many iterations can be promised by the classical oracle because the taciturn oracle is weaker in power.

1.1 Motivation

Unlike its counterpart under the classical oracle, IGS under the taciturn oracle has not been optimally solved. However, it is known [16] that the query cost for the taciturn oracle must be $\Omega(\log n + (d/k)\log_d n)$ in the worst case, implying that the algorithm of [16] can be unnecessarily slow by a $\log k$ factor (e.g., when $k \geq d$). Closing this gap was left as an open problem in [16]. We will settle the open prolem in this work.

There is a more fundamental issue from a methodological point of view. The above lower bound rules out the possibility of using the taciturn oracle to guarantee as few iterations as the classical oracle. Indeed, the cost $O(\log_k n + (d/k)\log_d n)$ for the classical oracle is strictly lower than the best possible performance $\Omega(\log n + (d/k)\log_d n)$ of the taciturn oracle. Is it possible to "combine the best of both worlds"? More specifically, can we discover another one-click oracle that (i) requires a human to click on one button per query and (ii) permits locating any target within $O(\log_k n + (d/k)\log_d n)$ queries? In this work, we will design a new oracle that offers an affirmative answer to the question.

Our final motivation stems from the somewhat surprising fact that previous work has not given due attention to CPU time. All the modern IGS algorithms rely on the so-called *heavy-path depth first search* (HPDFS) tree. Introduced in [22], this is a tree produced by a non-conventional depth-first search (DFS) — as will be reviewed in the next section — on the input DAG $G = (V, E)$ whose efficient computation is non-trivial. The fastest algorithm to this date requires $O(dnm)$ time [22], where $m = |E|$. In this work, we will provide a new algorithm to drastically reduce the time complexity.

1.2 Contributions

Our first contribution is an algorithmic framework that significantly simplifies the design of IGS algorithms. To apply our framework, one only needs to concentrate on two operations:

- Existence(Q): given a set Q of k vertices, return a binary answer indicating whether Q has at least one vertex capable of reaching the target.
- First-in-Order($\bar{Q}$): given a sequence $\bar{Q}$ of k vertices, return the first vertex in $\bar{Q}$ able to reach the target, or "none" if no vertex in $\bar{Q}$ can do so.

The astute reader would notice that Existence(Q) is subsumed by First-in-Order($\bar{Q}$) in functionality — so why do we need both? The answer lies in the fact that (i) our framework invokes the former operation more frequently than the latter, but (ii) the former may be cheaper than the latter. Indeed, the core of algorithm design is now reduced to figuring out how to leverage the given oracle to implement the two operations. If the first operation can be supported with cost T_E and the second operation can be supported with cost T_F, our framework automatically yields a concrete IGS algorithm with cost $O(\log_k n \cdot (T_E + T_F) + T_E \cdot (d/k)\log_d n)$.

With our framework in place, deriving IGS algorithms for various oracles becomes much easier. Consider the classical oracle first. As the oracle reveals the reachability (to the target) for every vertex of Q, it trivially supports both operations with costs $T_E = T_F = 1$. Accordingly, our framework achieves cost $O(\log_k n + (d/k)\log_d n)$, which matches the state of the art in [16].

Now, let us think about the taciturn oracle. First of all, the oracle does precisely what is needed for Existence(Q), giving $T_E = 1$. Furthermore, we can use the oracle to handle First-in-Order($\bar{Q}$) with binary search. Specifically, let $\bar{Q}_1$ (resp., $\bar{Q}_2$) be the sequence of the first (resp., last) $k/2$ vertices in $\bar{Q}$. Invoke the oracle to detect whether $\bar{Q}_1$ has a vertex that can reach the target. If so, recurse on $\bar{Q}_1$; otherwise, recurse on $\bar{Q}_2$. This ensures $T_F = \log_2(1 + k)$. Accordingly, our framework yields an algorithm with cost $O(\log n + (d/k)\log_d n)$. This is asymptotically optimal, strictly improves the state of the art [16], and closes the gap left behind by [16].

Finally, our framework naturally suggests a novel oracle:

> **First-in-order (FIO) oracle:** given a sequence $\bar{Q}$ of k vertices, the oracle returns the first vertex in $\bar{Q}$ able to reach the target, or "none" if no vertex in $\bar{Q}$ can do so.

This is another one-click oracle. For example, in the scenario of Figure 1, when given the sequence $\bar{Q}$ = (`vehicle`, `mammal`, `tiger`) and a picture of tiger, the human will click on `mammal` (i.e., the first applicable label in the sequence). Let us analyze the number of iterations that can be guaranteed by the FIO oracle. Clearly, the oracle does exactly what First-in-Order demands, meaning $T_F = 1$. To support Existence(Q), one may order Q into an arbitrary permutation $\bar{Q}$ and then invoke the oracle. The output for the operation is "yes" if and only if the oracle returns a vertex in $\bar{Q}$. Hence, T_E is also 1. As a result, our framework yields an IGS algorithm with cost $O(\log_k n + (d/k)\log_d n)$, which asymptotically matches the number of iterations that the classical oracle can ensure. Compared with the classical oracle, the FIO oracle reduces the mental effort because a human can now stop processing the given sequence once s/he sees the leftmost applicable option. With this, we combine the best of both worlds.

Our second contribution is a fast algorithm for computing the HPDFS tree. The novelty behind our algorithm is to cut the input graph $G = (V, E)$ into disjoint components using "bridges". Specifically, a *bridge* is an edge whose removal disconnects G into two disconnected subgraphs. By removing all the bridges from G, we decompose G into a set of subgraphs such that no edge exists between any two subgraphs. If Δ denotes the largest number of edges in a subgraph, our algorithm computes the HPDFS-tree in $O(m + n \cdot \Delta)$ time. In practice, the input graphs in IGS applications are sparse such that Δ is far less than m. Under such circumstances, our algorithm significantly improves the time complexity $O(dnm)$ of the current method.

We present a thorough empirical evaluation of (i) the existing and the proposed IGS algorithms and (ii) our new HPDFS computation algorithm. In all the scenarios inspected, the proposed IGS algorithms consistently outperform the existing ones; furthermore, our HPDFS algorithm is considerably (over an order of magnitude) faster than the existing implementations. Given the prevalence of large language models (LLMs), the last part of our experiments explores the potentials of integrating LLMs with IGS. We find that LLM-only methods exhibit low accuracy in identifying target nodes, indicating that the IGS algorithms in this paper are not subsumed by LLMs. Nevertheless, we demonstrate how LLMs can be used to reduce the amount of human effort needed by our IGS algorithms.

2 Preliminaries

This section will present a self-contained tutorial on the key concepts relevant to our technical discussion. Our examples will use the DAG $G = (V, E)$ in Figure 2 where V has 23 vertices and E includes all the solid and dashed edges. Vertex 1, which can reach every other vertex in the graph, is the root of G.

HPDFS-Trees. Depth first search (DFS) performs graph traversal using a stack. The algorithm pushes each vertex into the stack once and then pops it out once. The root of G is the first vertex entering the stack. At each step, if vertex u currently tops the stack, the algorithm looks for an out-neighbor v of u such that v has never been seen before. If such a vertex v exists, it is *discovered because of* u and is pushed into the stack. Otherwise, the algorithm pops u from the stack, and continues until the stack is empty. It is customary to use three colors to represent the status of each vertex: (i) **white**, if the vertex has not been discovered; (ii) **gray**, if the vertex has been seen and remains in the stack; (iii) **black**, if the vertex has been popped from the stack.

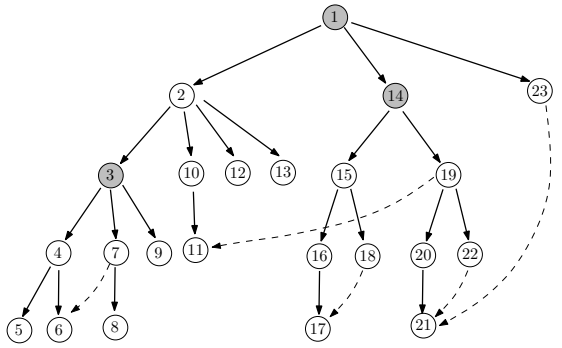


Fig. 2. A running example (solid edges make the HPDFS-tree, and the gray vertices make a 4-separator)

In traditional DFS, the vertex v discovered next can be an arbitrary white (i.e., unseen) out-neighbor of the vertex u that tops the stack. However, to compute a heavy-path depth-first search (HPDFS) tree — to be formalized shortly — we must choose v as the white out-neighbor of u having the highest wr-count (where "wr" means white reachability), defined as follows.

Definition 1. *The* **wr-count** *of a vertex v is the current number of white vertices that v can reach by going through only white vertices.*

Consider the graph G in Figure 2. In the beginning, the stack has only vertex 1, which is the only gray vertex, with all the other vertices being white. At this moment, the wr-count of vertex 2 equals 12 because it can reach vertices 2-13. Similarly, the wr-count of vertex 14 is 10 (it can reach vertices 11 and 14-22), and that of vertex 23 is 2 (it can reach vertex 21 and itself). Hence, vertex 2 is now discovered and enters the stack. Next, the algorithm pushes vertex 3 (with wr-count 7) into the stack, followed by vertices 4 and 5. Then, the algorithm pops vertex 5, pushes and then pops vertex 6, and pops out vertex 4. At this moment, vertices 4-6 are black, and the stack — from bottom to top — is (vertex) 1, 2, 3. Vertex 3 now has two white out-neighbors: vertex 7 whose wr-count is 2 (note that vertex 6 is black so does not contribute to the count), and vertex 9 whose wr-count is 1. Thus, vertex 7 is discovered next. The algorithm then continues in the same fashion.

We refer to the above special version of DFS as HPDFS.

Definition 2. *The* **HPDFS-tree** *of G is a tree where (i) the root is the root vertex of G and (ii) a vertex u parents vertex v if v is discovered because of u during HPDFS.*

Definition 3. *A vertex v has* **HPDFS-order** *r if it is the r-th vertex discovered during HPDFS.*

The solid edges shown in Figure 2 constitute the HPDFS-tree of our example DAG, while the vertex labels indicate the HPDFS-orders. For convenience, we order the children of every internal node from left to right in ascending order of HPDFS-order.

The HPDFS-tree is not easy to compute. The challenge is that every time a vertex is discovered, the wr-counts of many vertices may be affected. The fastest implementation today [22] re-computes, at each step of HPDFS, the wr-count for every white out-neighbor of the vertex u_{top} currently at the top of the stack. As each wr-count takes $O(m)$ time to compute and u_{top} has up to d out-neighbors, each step incurs an overhead of $O(d \cdot m)$ time. As the total number of steps is $n = |V|$ (there are n vertices to "discover"), the overall time complexity is $O(dnm)$.

Tree Separators, Left Flanks, and Stars. Henceforth, we will use T to denote the HPDFS-tree of $G = (V, E)$.

Definition 4. *For an integer $k \in [1, n]$, a set Σ of vertices in the HPDFS-tree T is a* **k-separator** *of T if (i) Σ includes the root of T, and (ii) removing the vertices of Σ (and their edges) disconnects T into connected components, each of which has at most $\lfloor n/k \rfloor$ vertices, where $n = |V|$.*

Lu et al. [16] proved that T always admits a k-separator containing at most k vertices, and this k-separator can be computed in $O(n)$ time. In Figure 2, the gray vertices 1, 3, and 14 constitute a 4-separator returned by their algorithm. Indeed, removing the three vertices breaks T into connected components, the largest of which has only $\lfloor 23/4 \rfloor = 5$ vertices.

Definition 5. *Let u be a vertex in the HPDFS-tree T. The* **left flank** *of u — denoted as $\mathrm{LF}(u)$ — is a set that includes the left siblings of every ancestor of u in T.*

The reader should note that (i) a node is as an ancestor of itself (as a convention in graph theory), and (ii) a node can have multiple left siblings. For example, in Figure 2, both vertices 2 and 14 are left siblings of vertex 23, giving $\mathrm{LF}(23) = \{2, 14\}$, where each number represents a vertex ID. As additional examples, $\mathrm{LF}(1) = \emptyset$, $\mathrm{LF}(9) = \{4, 7\}$, $\mathrm{LF}(18) = \{2, 16\}$, and $\mathrm{LF}(22) = \{2, 15, 20\}$.

Lu et al. [16] proved an interesting property of left flanks for HPDFS-trees. Let Σ be a k-separator of the HPDFS-tree T computed using their linear-time algorithm. Then, for any vertex $u \in \Sigma$, the size of $\mathrm{LF}(u)$ must be less than k. In Figure 2, as noted, $\Sigma = \{1, 3, 14\}$ is a 4-separator. We have: $\mathrm{LF}(1) = \mathrm{LF}(3) = \emptyset$, while $\mathrm{LF}(14) = \{2\}$. Clearly, all of them have sizes less than 4.

Definition 6. *Let T be the HPDFS-tree, and fix a target vertex $t \in V$. For any subset $S \subseteq V$, its* **star** *on t is the vertex $u \in S$ satisfying:*

(1) *u can reach t;*
(2) *no proper descendent of u in S can reach t;*
(3) *u has the smallest HPDFS-order among all the vertices in S satisfying conditions (1) and (2).*

In Figure 2, consider $S = \{1, 3, 14, 23\}$ and the target t is vertex 21. Vertex 3 violates condition (1). Vertex 1 satisfies (1), but violates (2) because it has a proper descendent — vertex 14 — that can reach t. Both vertices 14 and 23 satisfy conditions (1) and (2), but 14 has a smaller HPDFS-order. It follows that the star of S is 14.

3 A Generic IGS Algorithmic Framework

This section will describe our algorithmic framework for IGS. In Section 3.1, we will present two new observations about the notion of star defined in Section 2. Section 3.2 will review how IGS can be solved using the classical oracle, after which Section 3.3 will utilize our observations to develop an alternative IGS algorithm. Finally, in Section 3.4, we will provide an information-theoretic view on the inherent differences among different oracles.

3.1 New Properties of Stars

Let $G = (V, E)$ be the input DAG on which IGS is performed. Denote by T the HPDFS-tree of G from Definition 2. Echoing the HPDFS-order in Definition 3, next we introduce its "opposite":

Definition 7. *Suppose that, for each internal node in T, its child nodes have been arranged (from left to right) in ascending order of their HPDFS-orders. Then, the* **HPDFS-post-order** *of a vertex $u \in V$ is r if u is the r-th vertex visited in a post-order traversal of T.*

Recall that a post-order traversal recursively visits the i-th subtree of the root (with a post-order traversal) — in ascending order of i — before visiting the root. In Figure 2, a post-order traversal of the HPDFS-tree there lists vertices in this order: 5, 6, 4, 8, 7, 9, 3, 11, 10, 12, 13, 2, 17, 16, 18, 15, 21, 20, 22, 19, 14, 23, 1. The HPDFS-post-order of vertex 5, for instance, is 1, while that of vertex 11 is 8.

We are now ready to state our first observation:

Lemma 1. *Fix an arbitrary target $t \in V$. For any subset $S \subseteq V$, its star on t (Definition 6) is the vertex in S with the smallest HPDFS-post-order among those vertices in S able to reach t.*

Proof. To start with, note that, by enumerating the vertices of T in ascending order of HPDFS-order, one essentially performs a pre-order traversal of T. Recall that a pre-order traversal first visits the root, and then recursively visits the i-th subtree of the root with a pre-order traversal, in ascending order of i.

Denote by s^* the star of S on t. Let u be the vertex in S having the smallest HPDFS-post-order among those vertices in S able to reach t. If $s^* = u$, we are done.

Next, we assume $s^* \neq u$. By definition of star, no proper descendent of s^* can reach t; hence, u cannot be a proper descendent of s^*. On the other hand, as u has a smaller HPDFS-post-order than s^*, it follows that u cannot be a proper ancestor of s^*, either. In general, in any rooted tree, for two nodes without an ancestor-descendent relationship, the node with a smaller pre-order always has a smaller post-order. This means that node u must have a smaller HPDFS-order than s^*. However, in this case, s^* cannot satisfy condition (3) of Definition 6, giving a contradiction. □

As an illustration, consider the example of Figure 2 with $S = \{1, 3, 14, 23\}$ and the target $t = 21$. As mentioned in Section 2, the star of S on t is vertex 14. The above lemma offers a convenient way to derive this. First, notice that vertices 1, 14, and 23 are the only ones in S that can reach $t = 21$. Second, among those three vertices, 14 has the smallest HPDFS-post-order.

Our second observation has nothing to do with vertex ordering and is concerned with the specific subset $S = V$:

Lemma 2. *When $S = V$, the star of S on t is just the target t.*

Proof. Suppose that t is not the star. But why? Clearly, t satisfies condition (1) of Definition 6. Furthermore, it also satisfies condition (2), because otherwise there would be a cycle in the input graph G. Thus, t must violate condition (3), implying the existence of a node u that can reach t, does not have t as a proper descendent in T, and has a smaller HPDFS-order than t. However, this is impossible because the tree T has the following property (proved in [16]): if a node v_1 has a smaller HPDFS-order than another node v_2 but v_2 is not a proper descendent of v_1 in T, then v_1 cannot reach (in G) any vertex in the subtree of v_2 in T. Indeed, setting $v_1 = u$ and $v_2 = t$ violates the property. □

To "play with" the lemma, the reader may resort to Figure 2 and verify that, for any target t there, the star of V on t is always t.

3.2 The Classical Oracle

Recall that, given a set Q of k vertices, the classical oracle reveals for each vertex $u \in Q$ whether u can reach the target t. Next, we describe an algorithm of Lu et al. [16] — referred to as the *LMNT algorithm* following the authors' initials — that uses such an oracle to perform IGS. Later, we will integrate this algorithm with Lemmas 1 and 2 to design new algorithms. As before, let $G = (V, E)$ be the input DAG, and T be the HPDFS-tree of G. We will assume $k = 4$ in our subsequent examples.

If $\boldsymbol{|V| \leq k}$, the LMNT algorithm sets $Q = V$ and queries the oracle with Q. This reveals the reachability of every vertex to t and thus trivially finds the target.

Next, we consider the more typical situation $\boldsymbol{|V| > k}$, in which case the algorithm has two phases. In **phase 1**, it starts by finding a k-separator Σ of T and the star s^* of Σ on t. As mentioned in Section 2, Σ has at most k vertices; therefore, s^* can be found by issuing a single query to the oracle. For example, suppose that $t =$ (vertex) 11 in Figure 2. The 4-separator Σ that the LMNT algorithm finds is $\{1, 3, 14\}$. The oracle replies "yes", "no", and "yes" for those three vertices. With this, we can decide that s^* is vertex 14.

Phase 1 next retrieves the left flank of s^*, i.e., $\text{LF}(s^*)$. Now, the algorithm pinpoints the vertex s^{**} having the smallest HPDFS-order among the vertices in $\text{LF}(s^*) \cup \{s^*\}$ that can reach t. As noted in Section 2, the size of $\text{LF}(s^*)$ must be less than k. Hence, $\text{LF}(s^*) \cup \{s^*\}$ has at most k vertices, permitting s^{**} to be found with another query to the oracle. This concludes phase 1. In our earlier example, we have found $s^* = 14$. Thus, $\text{LF}(s^*) = \{2\}$. Given $Q = \{2, 14\}$, the oracle answers "yes" for both vertices, giving $s^{**} = 2$.

Phase 2 does nothing if $s^{**} \notin \Sigma$, and the algorithm simply renames s^{**} to $s^\#$. This is the case in our current example: $s^\# = 2$.

To demonstrate the opposite, let us switch to another target $t = 21$. By following the above steps of phase 1, one can verify that s^* and s^{**} are both vertex 14 this time. Thus, $s^{**} \in \Sigma$. In such case, the algorithm looks for the *leftmost* child of s^{**} in T that can reach t. If such a child does not exist, the target must be s^{**}. Otherwise, the phase finishes by setting $s^\#$ to the child. In our current example (with $t = 21$), $s^{**} = 14$ has child nodes 15 and 19. As vertex 15 cannot reach t, we have $s^\# = 19$.

In general, if $s^\#$ is the x-th child of s^{**}, then $s^\#$ can be found with $\lceil x/k \rceil$ queries (by dividing the child sequence into groups of size k and querying with each group in succession). If $s^\#$ does not exist, this can be detected with at most $\lceil y/k \rceil$ queries, where y is the number of children of s^{**} in T.

At the end of Phase 2, if t has not been found, the algorithm must be holding a vertex $s^\# \notin \Sigma$. It then collects a set $V^\#$ of vertices in the following procedure. Let T' be the subtree of $s^\#$ in T. For every node u in T' that falls in Σ, remove the subtree of u in T' from T'. Define $V^\#$ as the set of remaining nodes in T'. In our first example above with $t = 11$, $s^\# = 2$ and accordingly, $V^\# = \{2, 10, 11, 12, 13\}$, whereas in our second example with $t = 21$, $s^\# = 19$ and accordingly, $V^\# = \{19, 20, 21, 22\}$.

Finally, the algorithm extracts the subgraph $G^\# = (V^\#, E^\#)$ of G induced by $V^\#$, namely, $E^\#$ includes every edge of G whose both vertices are in $V^\#$. It then **recurses** by finding the target t in $G^\#$. As shown in [16], the entire recursion has $O(\log_k n)$ levels.

3.3 A Black Box IGS Algorithm

We now explain how to deploy the operations Existence and First-in-Order (Section 1.2) as black boxes to solve the IGS problem. Our strategy is to "translate" the LMNT algorithm using these operations, which is made possible by the two lemmas in Section 3.1.

When $|V| \leq k$. By Lemma 2, it suffices to find the star of the full vertex set V on the target t. This can be achieved using Lemma 1. Specifically, we arrange the vertices of V into a sequence $\bar{Q}$ in ascending order of their HPDFS-post-orders. The sequence $\bar{Q}$ has length $|V| \leq k$. We can thus run First-in-Order($\bar{Q}$) and return directly the vertex output by the operation.

When $|V| > k$. Let us discuss phase 1 of the LMNT algorithm. Recall that it first finds the star s^* of Σ, where Σ is the k-separator of T found. Equipped with Lemma 1, we sort the vertices of Σ in ascending order of HPDFS-post-order, and feed the sorted sequence $\bar{Q}$ into First-in-Order. The output of the operator is precisely s^*.

Phase 1 then finds the vertex s^{**} in $\text{LF}(s^*) \cup \{s^*\}$ that has the smallest HPDFS-order among those vertices in $\text{LF}(s^*) \cup \{s^*\}$ capable of reaching t. For this purpose, we sort $\text{LF}(s^*) \cup \{s^*\}$ in ascending order of HPDFS-order into a sequence $\bar{Q}$ and obtain s^{**} as the output of First-in-Order($\bar{Q}$).

Let us switch attention to phase 2. If $s^{**} \notin \Sigma$, then $s^\# = s^{**}$. Otherwise, we need to find the leftmost child $s^\#$ of s^{**} in T that can reach t, or declare that no such child exists. To do so, we chop the child list of s^{**} into $\lceil y/k \rceil$ groups — each containing k child nodes, except possibly the last one — where y is the number of children of s^{**} in T. Arrange these groups from left to right by respecting the original ordering of the child nodes. We process the groups with operation Existence according to that order: for each group, take the set Q of k vertices therein and perform Existence(Q). If the operation returns "no" for all the $\lceil y/k \rceil$ groups, we know that $s^\#$ does not exist, in which case t must

be s^{**} (as in the LMNT algorithm). Otherwise, we stop at the first group Q for which Existence(Q) returns "yes" — this must be the group containing $s^{\#}$. So far, $\lceil x/k \rceil$ Existence operations have been performed if $s^{\#}$ is the x-th child of s^{**}. If $\bar{Q}$ represents the sequence of child nodes in that group, we can now obtain $s^{\#}$ as the output of First-in-Order($\bar{Q}$).

This finishes one level of recursion. If t has not been located, then the algorithm recurses on $G^{\#}$, where $G^{\#}$ is a subgraph derived using $s^{\#}$ in the way described in Section 3.2.

Analysis. Denote by T_E (resp., T_F) the cost of applying the operation Existence (resp., Find-in-Order) once. When $|V| \leq k$, our framework incurs cost T_F as it performs only one First-in-Order operation. Next, we consider $|V| > k$.

Let us first account for the total cost of phase 1. At each recursion level, phase 1 performs Find-in-Order twice, which requires cost $2 \cdot T_F$. As mentioned in Section 3.2, the whole recursion has $O(\log_k n)$ levels. Hence, the overall cost from phase 1 is $O(T_F \cdot \log_k n)$.

It remains to bound the total cost of phase 2. Two observations are crucial. First, at each recursion level, phase 2 requires at most one First-in-Order. Thus, overall, the total cost of First-in-Order from phase 2 across all recursion levels is $O(T_F \cdot \log_k n)$. Second, at each recursion level, our algorithm performs Existence as many times as the number of queries issued by the LMNT algorithm in phase 2. Thus, the total number of Existence operations in our algorithm cannot exceed the total number of queries in the LMNT algorithm, which is $O(\log_k n + (d/k) \log_d n)$. This means that the overall cost of Existence is $O(T_E \cdot (\log_k n + (d/k) \log_d n))$.

Theorem 3. *The cost of our framework is $O(\log_k n \cdot (T_E + T_F) + T_E \cdot (d/k) \log_d n)$, where $n = |V|$, d is the maximum out-degree of G, and T_E (resp., T_F) is the cost of each* Existence *(resp.,* Find-in-Order*).*

3.4 Discussion

As discussed in Section 1.2, our framework yields asymptotically optimal IGS algorithms for all three oracles simultaneously: classical, taciturn, and FIO (first-in-order). In particular, the number of queries is $O(\log_k n + (d/k) \log_d n)$ for both the classical and FIO oracles, while the number is $O(\log n + (d/k) \log_d n)$ for the taciturn oracle. Intuitively, the number has to be higher for taciturn because this oracle reveals only one bit of information each time: simply a yes or no answer, i.e., whether the given set Q has a vertex that can reach the target t. In contrast, the classical oracle reveals k bits of information: one can think of its answer as a k-bit string, where the i-th bit ($1 \leq i \leq k$) indicates whether the i-th vertex in Q can reach the target t. On the other hand, the FIO oracle gives $\log_2(1 + k)$ bits of information each time: one can regard its answer as an integer from 1 to $k + 1$ — if the answer is $i \leq k$, it represents the i-th vertex in the given sequence $\bar{Q}$, while if the answer is $k + 1$, it means that no vertex in $\bar{Q}$ can reach t.

Besides producing the first optimal taciturn IGS algorithm, our framework also reveals — somewhat unexpectedly — the redundancy of the classical oracle. As one can see, even though it gives much more information than the FIO oracle, the latter suffices for achieving the same query complexity. The reader should recall from Section 1 that $\Omega(\log_k n + (d/k) \log_d n)$ is a lower bound for the classical oracle. The same lower bound obviously applies to the FIO oracle (which is *less* powerful) as well. This makes the FIO oracle particularly interesting because it is "one-click", as explained in Section 1.2. Given the above, we believe that our new algorithmic framework brings us closer to the essence of IGS.

4 Computation of the HPDFS-Tree

This section will focus on the problem of computing the HPDFS-tree of a DAG $G = (V, E)$ with a single root. The fastest known algorithm runs in $O(dnm)$ time [22], where $n = |V|$, $m = |E|$, and d

is the maximum vertex out-degree in G. We will improve this time complexity significantly based on several non-trivial ideas.

4.1 An $O(nm)$-Time Algorithm

We will describe a method to compute the HPDFS-tree in $O(nm)$ time. The method will be a building block in our final algorithm.

As explained in Section 2, HPDFS is a special DFS whose execution is guided by vertices' wr-counts. These wr-counts may change dynamically as the traversal progresses, and it would be exceedingly expensive to maintain their accurate values. Our first idea is to keep track of them only approximately until the moment they are actually needed.

At all times, we store for each vertex $u \in V$:

- a value $\alpha\text{-}cnt(u)$ as an upper bound of the wr-count of u.
- a field $best\text{-}out(u)$, which is the white out-neighbor of u having the smallest $\alpha\text{-}cnt$. If u has no white out-neighbors, then $best\text{-}out(u) = \emptyset$.

In addition, we enforce:

> **Just-in-time accuracy**: When HPDFS looks for the next vertex to visit, the following requirement needs to be satisfied. Let u_{top} be the vertex at the top of the stack. For any white out-neighbor v of u_{top}, the value of $\alpha\text{-}cnt(v)$ should equal the current wr-count of v.

The above policy ensures that the next vertex to visit is simply $best\text{-}out(u_{\text{top}})$, namely, HPDFS will push the vertex $best\text{-}out(u_{\text{top}})$ into the stack and turn it from white to gray. On the other hand, if $best\text{-}out(u_{\text{top}}) = \emptyset$, then u_{top} will be popped from the stack and turn from gray to black.

At the beginning of HPDFS, we set, for each vertex $u \in V$, the value $\alpha\text{-}cnt(u)$ to how many vertices are reachable from u in G (note: u can reach itself). This can be obtained in $O(m)$ time by performing a conventional DFS on G from u. Thus, the $\alpha\text{-}cnt$ values of all vertices can be initialized in $O(nm)$ time, after which all the $best\text{-}out$ fields can be decided easily in $O(m)$ total time.

During HPDFS, we update the $\alpha\text{-}cnt$ and $best\text{-}out$ values only when a vertex v turns black, or equivalently when v gets popped from the stack. When this happens, we perform a so-called **reverse update**, which

- first finds the set S of vertices in G able to reach v, and
- then decreases $\alpha\text{-}cnt(u)$ by 1 for each vertex $u \in S$.

The set S can be gathered in $O(m)$ time by a conventional DFS from v on the *reverse* graph of G (as is generated by reversing the direction of every edge in G). After that, the $best\text{-}out$ fields of all vertices can be updated accordingly in $O(m)$ total time. Every vertex turns black exactly once during the whole HPDFS. Therefore, n reverse updates are required in total, with an overall cost of $O(nm)$.

Let us illustrate reverse updates using Figure 2. As discussed in Section 2, vertex 5 turns black first, at which point the stack contains (from bottom to top) vertices 1, 2, 3, and 4. The reverse update decreases the $\alpha\text{-}cnt$ values of vertices 1-5. Vertex 6, turning black second, triggers a reverse update that reduces the $\alpha\text{-}cnt$ values of vertices 1-4, 6, and 7. Now, $\alpha\text{-}cnt(7) = 2$. Next, when vertex 4 turns black, it further lowers the $\alpha\text{-}cnt$ values of vertices 1-4. Now, vertex 3 tops the stack, and $best\text{-}out(3) = 7$ because $\alpha\text{-}cnt(7) = 2 > 1 = \alpha\text{-}cnt(9)$. The value of $\alpha\text{-}cnt(7)$ equals exactly the wr-count of vertex 7, as promised by just-in-time accuracy. HPDFS then pushes vertex 7 into the stack and continues.

To establish our algorithm's correctness, it remains to prove:

Lemma 4. *The just-in-time accuracy policy is enforced after each reverse update.*

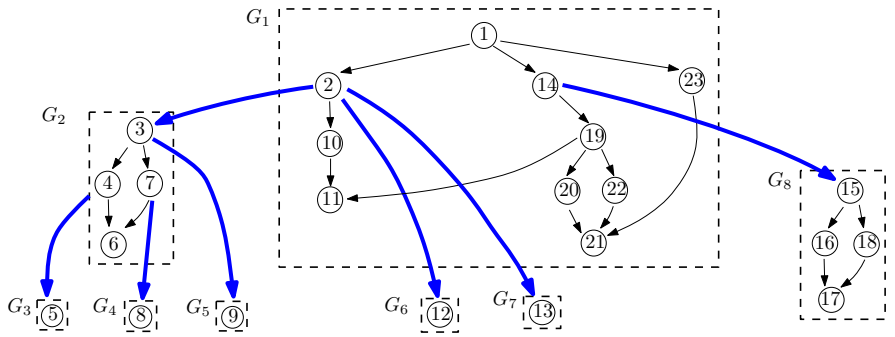


Fig. 3. Bridges (in blue) and the effects of their removal

Proof. Consider the moment after a reverse update. Let u_{top} be the vertex at the top the stack, and v be any of its white out-neighbors. Our algorithm ensures that $\alpha\text{-}cnt(v)$ is exactly the number of vertices u in G such that (i) v can reach u, and (ii) the color of u is not black. Notice that the color of such a vertex u cannot be gray; otherwise, u is in the stack and thus has a path to v, indicating the presence of a cycle. Hence, u must be white. Next, we argue that v can reach u via a path of white vertices. This indicates that $\alpha\text{-}cnt(v)$ equals the wr-count of v, as claimed.

Consider an arbitrary path π from v to u in G. We claim that every vertex on π must be white. Otherwise, let u' be the non-white vertex on π nearest to u. First, u' cannot be gray; otherwise, a cycle would exist because u' would be in the stack and hence could reach v. Hence, u' must be black. However, the well-known *white-path theorem* [5] — which applies to any DFS and hence also HPDFS — states that, before u' turns black, DFS must have visited all vertices reachable from u'. This contradicts the fact that u is still white. □

We conclude that the HPDFS-tree can be found in $O(nm)$ time.

4.2 Bridges

Let us start with a definition:

Definition 8. *An edge $G = (V, E)$ is a* **bridge** *if its removal disconnects G into two subgraphs between which no edges exist.*

In general, if b is the number of bridges in the input DAG G, their removal breaks G into $b + 1$ subgraphs such that no edges exist between any two of those subgraphs (this implies $b < n$). Consider Figure 2 again. There are $b = 7$ bridges: (2, 3), (4, 5), (7, 8), (3, 9), (2, 12), (2, 13), (14, 15). To illustrate these subgraphs, in Figure 3, we display each of them in a box and rearrange their positions into a hierarchy where different boxes are connected using bridges (shown in blue).

Let us represent the $b + 1$ subgraphs mentioned earlier as G_1, G_2, ..., G_{b+1}, respectively. By viewing each subgraph as a "super vertex", these subgraphs form a tree, which we denote as $\mathcal{T}_{\text{super}}$. Specifically, a subgraph G_i *parents* another subgraph G_j if there is a bridge (u, v) going from a vertex u in G_i to a vertex v in G_j. Conversely, G_j is said to be a *child* of G_i. The vertex u is an **exit vertex** of G_i, and the vertex v is an **entry vertex** of G_j. All the standard tree terminology applies. For example, a subgraph $G_{i'}$ is a *descendent* of another subgraph $G_{j'}$ if $G_{j'}$ is in the subtree of $G_{i'}$ in T_{super}. Furthermore, it is a *proper descendent* of $G_{j'}$ if $i' \neq j'$.

In Figure 3, for example, G_1 is the parent of G_2, G_6, G_7, and G_8. Vertices 2 and 14 are exit vertices of G_1, while 3, 12, 13, and 15 are the entry vertex of G_2, G_6, G_7, and G_8, respectively. All the subgraphs G_2, ..., G_7 are proper descendent subgraphs of G_1.

Any DFS — HPDFS included — guarantees:

One-way property: Consider any bridge (u, v) and let G_i (for some $i \in [1, b]$) be the subgraph of T_{super} containing v. Once v is discovered, the DFS will finish visiting all the descendent subgraphs of G_i before popping v from the stack.

Let us familiarize ourselves with how HPDFS runs from the hierarchy perspective in Figure 3. After visiting vertices 1 and 2, the algorithm discovers vertex 3 via a bridge. After that, the algorithm will first visit all the vertices in G_2, G_3, G_4, and G_5 before backtracking to vertex 2. Furthermore, this one-way property applies recursively. For instance, in this example, after discovering vertex 3, HPDFS visits vertex 4 and then crosses a bridge to reach vertex 5. The property states that the algorithm will finish exploring G_3 first before backtracking to vertex 4.

All the bridges of G can be found in $O(m)$ time [23], after which it is rudimentary to obtain the resulting subgraphs $G_1, G_2, ..., G_{b+1}$ in another $O(m)$ time. For each $i \in [1, b+1]$, we use V_i (resp., E_i) to denote the set of vertices (resp., edges) in G_i, and define $n_i = |V_i|$ and $m_i = |E_i|$. Define $\Delta = \max_{i=1}^{b+1} m_i$, namely, the maximum number of edges in a subgraph.

4.3 An $O(m + n \cdot \Delta)$-Time Algorithm

This section will present an algorithm to compute the HPDFS-tree of G in $O(m + n \cdot \Delta)$ time. Our starting point is still the $O(nm)$-time algorithm — henceforth referred to as the *base method* — in Section 4.1. However, we will implement the method in a faster way by resorting to the subgraph hierarchy T_{super}.

Computing Initial α-cnt Values. Recall that the base method starts by computing, for each vertex $u \in V$, an integer $\alpha\text{-}cnt(u)$ equal to the number of vertices reachable from u in G. Next, we will explain how to finish such computation in $O(m + n \cdot \Delta)$ time.

First, arrange the subgraphs in a *reverse topological order* where a child subgraph always precedes the parent subgraph. Such orders are not unique, and we can find one in $O(m)$ time by doing a post-order traversal on T_{super}. In Figure 3, for example, such a traversal yields the order: $G_3, G_4, G_5, G_2, G_6, G_7, G_8, G_1$. We will process the subgraphs according to the reverse topological order. When processing a subgraph G_i (for some $i \in [1, b+1]$), we make the inductive assumption that $\alpha\text{-}cnt(v)$ has already been properly computed for every vertex v in all the proper descendent subgraphs of G_i. This is trivially true if G_i is a leaf in the hierarchy T_{super}.

To explain the processing of $G_i = (V_i, E_i)$, let us first define a $\beta\text{-}cnt(u)$ value for every exit vertex $u \in V_i$:

$$\beta\text{-}cnt(u) = \sum_{\text{bridge } (u,v)} \alpha\text{-}cnt(v). \tag{1}$$

The definition is valid because the vertex v in a bridge (u, v) must be in a proper descendent subgraph of G_i, and hence its $\alpha\text{-}cnt(v)$ is readily available based on our inductive assumption. Furthermore, the value $\beta\text{-}cnt(u)$ has an important meaning: it is how many vertices u can reach in the proper descendent subgraphs of G_i. The time of deriving the β-cnt values for all exit vertices of V_i is $O(1 + \#\text{ bridges leaving } G_i)$.

Consider, for example, the exit vertex 2 of G_1 in Figure 3. When G_1 is processed, $\alpha\text{-}cnt(3) = 7$, $\alpha\text{-}cnt(12) = 1$, and $\alpha\text{-}cnt(13) = 1$ are all ready. Then, $\beta\text{-}cnt(2) = 7 + 1 + 1 = 9$, which is precisely how many vertices that vertex 2 can reach in G_2, G_3, G_4, G_5, G_6, and G_7.

Now, we compute the value of $\alpha\text{-}cnt(z)$ for each vertex $z \in V_i$. First, find the set S of vertices that z can reach in G_i. Then, set:

$$\alpha\text{-}cnt(z) = |S| + \sum_{\text{exit vertex } u \in S} \beta\text{-}cnt(u). \tag{2}$$

The summation on the right hand side of (2) gives the number vertices reachable from z in the proper descendent subgraphs of G_i. As the set S can be acquired by running DFS from z within G_i, the cost of calculating α-$cnt(z)$ is $O(m_i)$ where $m_i = |E_i|$. Doing so for all the $z \in V_i$ takes $O(n_i m_i)$ time, where $n_i = |V_i|$.

In Figure 3, subgraph G_1 has two exit vertices: 2 and 14, with β-$cnt(2) = 9$ and β-$cnt(14) = 4$. To compute α-$cnt(1)$, we first get $S = \{1, 2, 10, 11, 14, 19\text{-}23\}$. Thus, (2) yields α-$cnt(1) = 10 + 9 + 4 = 23$.

Using the above strategy, we compute the α-cnt values for all the vertices of G in

$$\sum_{1 \leq i \leq b+1} O(n_i m_i + \#\text{ bridges leaving } G_i) \tag{3}$$

time. Note that $\sum_{i=1}^{b+1}(\#\text{ bridges leaving } G_i)$ is exactly b, i.e., the total number of bridges. On the other hand, applying $m_i \leq \Delta$ for all $i \in [1, b+1]$, we have $\sum_{i=1}^{b+1} n_i m_i \leq \Delta \cdot \sum_{i=1}^{b+1} n_i = n \cdot \Delta$. Thus, the cost in (3) is $O(m + n \cdot \Delta)$ (recall that $b < n = O(m)$).

Out-Neighbor Separation for Exit Nodes. As mentioned, the α-cnt value of each vertex u at this moment equals how many vertices are reachable from u in the whole G. We will remember this value as r-$cnt(u)$. One can regard r-$cnt(u)$ as a "snapshot" of α-$cnt(u)$ because the former will not change in the rest of the algorithm, unlike the latter.

Consider u as an exit vertex; denote by $G_i = (V_i, E_i)$ the subgraph that u belongs to. We define an out-neighbor v of u as **internal** if $v \in V_i$ and as **external** otherwise (this means (u, v) must be a bridge). We collect all the external out-neighbors v of u and sort them in descending order of r-$cnt(v)$ to form what we call the *external out-neighbor list* (**EON-list**) of u. For example, in Figure 3, the EON-list of the exit vertex 2 contains (in this order) 3, 12, and 13, whose r-cnt values are 7, 1, and 1, respectively.

Naively, we could obtain the EON-lists for all the vertices in $O(n \log d)$ total time, but this would make our final time complexity $O(m + n \cdot \Delta + n \log d)$. To avoid the $O(n \log d)$ term, we utilize counting sort to produce all the EON-lists in $O(m)$ time. A crucial observation is that every r-cnt value is an integer at most n. Thus, counting sort allows us to arrange all the $\overline{\text{entry}}$ vertices $v \in V$ in descending order of r-$cnt(v)$ using $O(n)$ time. After that, we process each $\overline{\text{entry}}$ vertex v in the sorted order and append it to the EON-list of each vertex u satisfying the condition that u is an exit vertex and also an *in-neighbor* of v. The total cost is proportional to the total in-degree of all the entry vertices, which is at most m. In Figure 3, the sorted list of entry vertices is: 3, 15, 5, 8, 9, 12, 13 where r-$cnt(3) = 7$, r-$cnt(15) = 4$, and all other vertices have r-cnt value 1. The processing of 3, 15, 5, 8, 9, 12, and 13 appends the vertex to the EON-list of vertex 2, 14, 4, 7, 3, 2, and 2, respectively.

Reverse Updates. The remaining task is to implement the reverse updates in the base method of Section 4.1. Recall that the purpose of such updates is to maintain the α-cnt and *best-out* fields for all the vertices in V. In the base method, whenever a vertex v turns black, we decrease α-$cnt(u)$ for every vertex u that can reach v. We can no longer afford this if the goal is to ensure running time $O(m + n \cdot \Delta)$.

As far as the just-in-time accuracy policy (Section 4.1) is concerned, we can delay many changes made by reverse updates by harnessing the one-way property (Section 4.2). To explain the rationale, let us run HPDFS again on the DAG in Figure 3. After pushing vertices 1 and 2 into the stack, HPDFS discovers vertex 3 by crossing the bridge $(2, 3)$. The one-way property assures us that the traversal will be "trapped" in the descendent subgraphs of G_2 until all the vertices therein have turned black. Let v be any vertex in G_2, G_3, G_4, or G_5. When v turns black, the base method decreases (among others) the values of α-$cnt(2)$ and α-$cnt(1)$. Hence, by the time HPDFS backtracks from the bridge $(2, 3)$ — popping out vertex 3 from the stack — α-$cnt(2)$ and α-$cnt(1)$ have both been decreased precisely 7 times, where 7 is the total number of vertices in G_2, G_3, G_4, and G_5, and is

also the *r-cnt* value of the entry vertex 3! Therefore, rather than modifying α-$cnt(2)$ and α-$cnt(1)$ every time a vertex v turns black, we can skip those modifications altogether. Instead, when HPDFS backtracks from the bridge (2, 3), we can find all the vertices in G_1 capable of reaching vertex 2 — that is vertices 1 and 2 — and for every such vertex u, decrease its α-$cnt(u)$ by r-$cnt(3) = 7$.

Formally, every time a vertex v turns black, we perform a reverse update only in the subgraph, say G_i, that contains v. Specifically, we find every vertex z in G_i that can reach v — this requires a conventional DFS on the reverse graph of G_i and finishes in $O(m_i) = O(\Delta)$ time — and decrease α-$cnt(z)$ by 1. On the other hand, every time we backtrack from a bridge (u, v) (i.e., popping out v while having u at the top of the stack) — assuming that u is in subgraph G_j — we do an **aggregate reverse update** in G_j, which

- finds every vertex z in G_j able to reach u, and
- decreases α-$cnt(z)$ by r-$cnt(v)$.

This aggregate reverse update requires a conventional DFS on the reverse graph of G_j and finishes in $O(m_j) = O(\Delta)$ time.

Let us illustrate the above using Figure 3. HPDFS first visits vertices 1-5 and then pops out 5, inducing a reverse update in G_3 that lowers α-$cnt(5)$ to 0. The traversal backtracks to vertex 4, triggering an aggregate reverse update in G_2 to reduce α-$cnt(4)$ and α-$cnt(3)$ by r-$cnt(5) = 1$. The algorithm continues to vertex 6, which then turns black, causing a reverse update in G_2 from vertex 6 to decrease α-$cnt(6)$, α-$cnt(4)$, α-$cnt(3)$, and α-$cnt(7)$ by 1. Next, HPDFS pops 4, pushes and pops vertices 7, 8, and 9; each pop is followed by a reverse update or an aggregate reverse update in G_2. Now, vertex 3 is popped. Backtracking from vertex 3, the algorithm does an aggregate reverse update in G_1 from 2, reducing α-$cnt(2)$ and α-$cnt(1)$ by r-$cnt(3) = 7$. The rest execution is similar.

It remains to explain how to maintain the *best-out* fields. Recall that the base method does so in $O(m)$ time after each reverse update. Our goal here is to restore these fields after every reverse update and every aggregate reverse update, but we must reduce the restoration time from $O(m)$ to $O(\Delta)$. The complication is that the budget $O(\Delta)$ allows us to inspect only the edges *within* the subgraph where the update occurs. This poses an issue because the restoration of *best-out*(u) for an exit vertex u requires inspecting edges outside of the subgraph. To see why, consider the moment when HPDFS backtracks from the bridge $(2, 3)$ to vertex 2. Recall that we perform an aggregate reverse update from vertex 2 in subgraph G_1. To update *best-out*(2) at this point, we need the α-$cnt(v)$ value for every white out-neighbor v of vertex 2, that is, vertices 10, 12, and 13. However, identification of 12 and 13 is through the bridges $(2, 12)$ and $(2, 13)$, which do not belong to G_1.

To handle the issue, we separate the internal out-neighbors of an exit vertex u from its external ones. Specifically, two extra fields are maintained for u:

- *best-int-out*(u): the white internal out-neighbor of u with the largest α-*cnt* value currently;
- *best-ext-out*(u): the white external out-neighbor of u with the largest α-*cnt* value currently.

Clearly, *best-out*(u) is merely the one between *best-int-out*(u) and *best-ext-out*(u) having a larger α-*cnt*. Consider a reverse update (aggregate or not) in subgraph G_i. For every exit vertex u in G_i, we restore *best-int-out*(u) by examining all its internal out-neighbors explicitly. As those out-neighbors are identified through edges in G_i, doing so for all exit vertices of G_i incurs at most a cost proportional to the number of edges in G_i, which is bounded by $O(\Delta)$.

How about *best-ext-out*(u)? It can be obtained in $O(1)$ time per exit vertex u! This is where the EON-list of u comes in. Due to the one-way property (Section 4.2), HPDFS must visit the external out-neighbors v of u in descending order of r-$cnt(v)$. Recall that each EON-list has been sorted in descending order of *r-cnt* values. Hence, *best-ext-out*(u) is simply the first vertex v' on the EON-List of u that is still white. When v' turns black, we can update *best-ext-out*(u) in $O(1)$ time by setting it to the next vertex on the list.

dataset	$n = \|V\|$	$m = \|E\|$	Δ
Amazon	29240	29239	0
ImageNet	27714	28190	760
WordNet	82115	84427	7162

(a) The n, m, and Δ values

level	0	1	2	3	4	5	6	7	8	9
avg out-degree	84	11	4.6	2.4	1.0	0.3	0.2	0.1	0.1	0
max out-degree	84	225	90	49	78	27	14	14	2	0

(b) Degree statistics of Amazon

level	0	1	2	3	4	5	6	7	8	9	10	11	12
avg out-degree	8	83	3.4	2.2	1.4	0.9	0.7	0.6	0.5	0.5	0.7	0.4	0
max out-degree	8	402	173	357	304	123	87	31	24	54	21	12	0

(c) Degree statistics of ImageNet

level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
avg out-degree	3	7.3	10	8.9	3.1	2.0	1.6	0.8	0.8	0.7	0.6	0.6	0.6	0.6	0.5	0.8	0.5	0.2	0
max out-degree	3	8	37	402	150	372	398	320	664	304	123	87	27	24	54	21	12	8	0

(d) Degree statistics of WordNet

Table 1. Real data statistics

Let us demonstrate the effect of separation using Figure 3. As mentioned, the EON-list of vertex 2 has (in this order) vertices 3, 12, and 13. In the beginning, *best-ext-out*$(2) = 3$ and *best-int-out*$(2) = 10$. After HPDFS backtracks from the bridge $(2, 3)$, *best-ext-out*(2) changes to 12. Recall that at this point an aggregate reverse update is carried out in G_1 from vertex 2, but *best-int-out*(2) still remains 10 after that. As α-*cnt*$(10) = 2 > 1 = \alpha$-*cnt*(3) currently, we are sure that *best-out*$(2) = 10$.

To analyze the overall running time, first notice that every vertex (when it turns black) triggers a reverse update in the subgraph where it belongs. Hence, the total number of reverse updates is n. Further notice that every entry vertex v (when it turns black) triggers an aggregate reverse update in the parent subgraph of the subgraph containing v. Hence, the total number of aggregate reverse updates is b (i.e., the number of bridges). As each reverse update — aggregate or not — takes $O(\Delta)$ time, we conclude that all these updates can be completed in $O((n + b) \cdot \Delta) = O(n \cdot \Delta)$ time.

We have proved the second main result of this paper:

Theorem 5. *Let $G = (V, E)$ be a DAG having a single root. Imagine removing all the bridges (Definition 8), which disconnects G into disjoint subgraphs. Let Δ be the maximum number of edges in a subgraph. Then, the HPDFS-tree of G can be computed in $O(m + n \cdot \Delta)$ time where $n = |V|$ and $m = |E|$.*

5 Experiments

Section 5.1 will describe the data used in our empirical evaluation. Then, Section 5.2 will inspect the interaction behavior of IGS algorithms, while Section 5.3 will examine their CPU efficiency. Finally, Section 5.4 will compare IGS to large language models (LLM) in a concrete application of categorization.

5.1 Data

Real Data. Our experiments deployed three real-world datasets. The first two — Amazon and ImageNet — have been used in [16, 22] previously. Amazon is a tree representing a product hierarchy at Amazon, while ImageNet is a (non-tree) DAG representing an annotation ontology for image categorization. The third real dataset — named WordNet[1] — was derived from an English lexical database, where words are grouped into "synsets", each being a set of synonyms. To generate a graph $G = (V, E)$, we extracted all the noun synsets as vertices in V, and created an edge in E

[1]Download at https://wordnetcode.princeton.edu/3.0/WNdb-3.0.tar.gz.

from a synset u to another synset v whenever u generalizes[2] v. The G obtained in this manner is a (non-tree) DAG.

Table 1a gives the values of $n = |V|$, $m = |E|$, and Δ for each dataset. Recall that Δ is the maximum number of edges in a subgraph after removing all the bridges; note that Δ is 0 for `Amazon` because this DAG is a tree.

We define the *level* of a vertex u in a (single-rooted) DAG G as the length of the shortest path from the root of G to u. The out-degree statistics per level for `Amazon`, `ImageNet`, and `WordNet` can be found in Tables 1b-d.

Synthetic Data. We also created synthetic data to study the influence of various control parameters on the behavior of algorithms. Each synthetic DAG $G = (V, E)$ was generated based on three parameters: (i) $n = |V|$; (ii) an integer d equal to the maximum out-degree in G; (iii) a real value $r \in [0, 1)$ that decides how much G deviates from a tree. The generation involves two parts: part I produces a tree T_{init}, whereas part II turns T_{init} to a DAG. Specifically, T_{init} is a tree of n nodes satisfying:

(1) The root has d children.
(2) Let h be the level of the deepest leaf of T_{init}. Level $i \in [1, h-1]$ has exactly $d \cdot f^{i-1}$ nodes, where $f = \lceil d \cdot (1-r) \rceil$. Level h has $n - 1 - \sum_{i=1}^{h-1}(d \cdot f^{i-1})$ nodes.
(3) With at most a single exception, every non-root internal node has f child nodes. The exception node (if exists), which may have less than f child nodes, must be the rightmost internal node at level $h-1$.

Part II converts T_{init} to a DAG by adding "cross edges". Specifically, for each non-root internal node u, we carry out two steps:

(1) If i is the level of u, we uniformly sample a set S of $d - f$ nodes at level $i+1$ that are not children of u in T_{init}.
(2) Add an edge from u to every node $v \in S$; we call this a *cross edge* because v is not a child of u. The out-degree of u becomes d after this step.

Henceforth, r will be referred to as the *cross ratio*. Note that the value of $d - f$ is roughly $d \cdot r$. When $r = 0$, no cross edges exist such that G is simply T_{init}. As r grows, the percentage of cross edges in G escalates accordingly, making G increasing "non-tree".

Unless otherwise stated, the parameters n, d, and r are set to their default values 1000000, 30, and 0.1, respectively.

5.2 Evaluation of Interaction Effectiveness

This subsection will evaluate IGS algorithms' performance in interacting with a human that plays the role of oracle, assuming that the oracle never errs[3].

Competing Methods. We studied four algorithms:

- `classical`: the asymptotically optimal algorithm of [16] for the classical oracle (introduced in Section 1).
- `LMNT-taciturn`: the algorithm of [16] for the taciturn oracle (introduced in Section 1).
- `new-taciturn`: the algorithm obtained by instantiating our framework in Theorem 3 with the implementation explained in Section 1.2 for the taciturn oracle.

[2]Namely, the concept represented by u is a hypernym of that of v.
[3]It is standard to tackle human errors through repetition on the same question. Assuming a worker has a probability of $c > 1/2$ answering a question correctly, taking the majority answer (on the same question) from a constant number of independent workers boosts the accuracy probability to 99.9% (the constant depends on c).

k	FIO/classical avg (max)	new-tacit avg (max)	LMNT-tacit avg (max)
1	26 (228)	26 (228)	26 (228)
2	14 (115)	18 (116)	19 (116)
4	9.1 (58)	16 (61)	19 (61)
6	7.6 (40)	17 (43)	18 (43)
8	7.7 (31)	17 (35)	18 (35)
10	7 (25)	17 (28)	19 (31)

(a) Number of queries vs. k on Amazon

k	FIO avg (max)	classical avg (max)
1	26 (228)	26 (228)
2	14 (115)	19 (117)
4	9.1 (58)	15 (60)
6	7.6 (40)	15 (43)
8	7.7 (31)	17 (35)
10	7 (25)	17 (29)

(b) Number of clicks vs. k on Amazon

k	FIO/classical avg (max)	new-tacit avg (max)	LMNT-tacit avg (max)
1	35 (402)	35 (402)	35 (402)
2	19 (201)	24 (203)	35 (203)
4	12 (102)	20 (104)	24 (105)
6	9.9 (69)	20 (71)	23 (73)
8	8.9 (52)	19 (56)	23 (58)
10	8 (42)	19 (48)	22 (52)

(c) Number of queries vs. k on ImageNet

k	FIO avg (max)	classical avg (max)
1	35 (402)	35 (402)
2	19 (201)	25 (203)
4	12 (102)	22 (106)
6	9.9 (69)	20 (73)
8	8.9 (52)	19 (56)
10	8.0 (42)	19 (46)

(d) Number of clicks vs. k on ImageNet

k	FIO/classical avg (max)	new-tacit avg (max)	LMNT-tacit avg (max)
1	46 (676)	46 (676)	46 (676)
2	25 (339)	30 (343)	34 (347)
4	16 (172)	26 (178)	30 (185)
6	12 (114)	24 (121)	31 (133)
8	11 (86)	23 (96)	30 (106)
10	9.6 (70)	23 (80)	32 (92)

(e) Number of queries vs. k on WordNet

k	FIO avg (max)	classical avg (max)
1	46 (676)	46 (676)
2	25 (339)	34 (347)
4	16 (172)	28 (182)
6	12 (114)	24 (122)
8	11 (86)	23 (95)
10	9.6 (70)	22 (79)

(f) Number of clicks vs. k on WordNet

Table 2. Number of queries/clicks vs. k (real data)

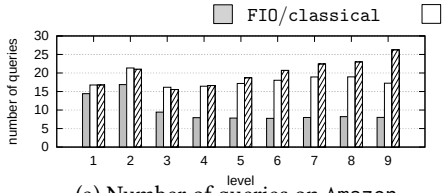
- FIO: the algorithm obtained by instantiating Theorem 3 with the implementation given in Section 1.2 for the FIO oracle.

Workloads. Given a DAG $G = (V, E)$, a *workload* on G is a subset $S \subseteq V$ such that each vertex in S defines an instance of IGS using that vertex as the target. A *full workload* comprises all the leaves of G (a leaf is a vertex with out-degree 0), whereas a *random workload* consists of 1000 random leaves in G. Our experiments always process a full workload on a real-world DAG and a random workload on a synthetic DAG (due to its vast size).

Metrics. For each algorithm, we measure

- the number of queries issued to the oracle;
- the number of (mouse) clicks by a human acting as the oracle.

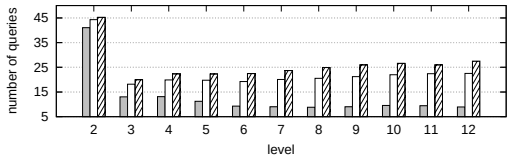
Recall that taciturn and FIO are both one-click oracles, namely, each query requires a human (oracle) to do only one click. The classical oracle — if implemented naively — would require a human to do $k = |Q|$ clicks per query, where Q is the vertex set presented by the query. We propose a better implementation, motivated by the fact that, in reality, most vertices in G cannot reach the target t anyway. Given a set Q of k vertices, the human is given a default choice of "no" for each vertex in Q. S/he clicks on a vertex $u \in Q$ if vertex u can reach t. If s/he has clicked on all k vertices, the algorithm automatically realizes that the human is done answering the query. However, if not all the vertices in Q can reach t, the human needs to click on a special "done" button to explicitly indicate the completion of a query. In this way, if x vertices in Q can reach t, the human performs $\min\{k, x + 1\}$ clicks.



(a) Number of queries on Amazon

level	1	2	3	4	5	6	7	8	9
FIO	14	17	9.4	7.9	7.8	7.8	8	8.2	8
classical	15	19	14	14	16	17	19	20	21

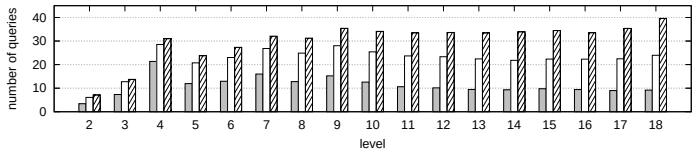
(b) Number of clicks on Amazon



(c) Number of queries on ImageNet

level	2	3	4	5	6	7	8	9	10	11	12
FIO	41	13	13	11	9.3	9	8.8	9	9.5	9.4	9
classical	45	19	21	20	20	20	21	22	23	22	23

(d) Number of clicks on ImageNet

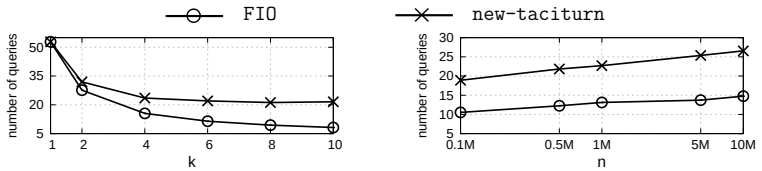


(e) Number of queries on WordNet

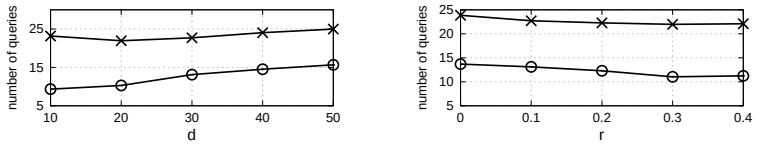
level	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
FIO	3.4	7.3	21	12	13	16	13	15	13	11	19	9.4	9.3	9.7	9.4	9	9.2
classical	6.5	13	28	21	23	27	25	29	26	25	25	25	24	25	24	24	26

(f) Number of clicks on WordNet

Fig. 4. IGS performance by level (real data, $k = 5$)



(a) Vs. k ($n = 1$M, $d = 30$, $r = 0.1$), (b) Vs. n ($k = 5$, $d = 30$, $r = 0.1$)



(c) Vs. d ($k = 5$, $n = 1$M, $r = 0.1$) (d) Vs. r ($k = 5$, $n = 1$M, $d = 30$)

Fig. 5. Number of queries — also number of clicks — for one-click oracles (synthetic data)

Results on Real Data. For each value $k \in [1, 10]$, we applied every competing method to execute a full workload S on every real dataset and measured (i) the average number of queries per IGS instance from S, and (ii) the maximum number of queries of all instances. The results are shown in Tables 2a, 2c, and 2e (with maximum numbers in brackets). The results of FIO and classical are presented together as they always issue exactly an identical number of queries.

All methods entailed the same cost at $k = 1$ because they degenerate into the same algorithm at $k = 1$. However, for $k > 1$, the FIO and classical oracles demanded much fewer queries than the taciturn oracle. Between the two taciturn algorithms, `new-taciturn` consistently outperformed `LMNT-taciturn`, with the highest improvement ratio reaching 23.3% (observed at $k = 10$ on `WordNet`). As an interesting observation, for `FIO` and `classical`, the maximum cost (of a workload) decreased almost linearly with k and exhibited a strong correlation with the maximum out-degree of the underlying DAG (as can be seen from Tables 1b-d).

Let us now attend to the number of clicks required by each method. For `FIO`, `new-taciturn`, and `LMNT-taciturn`, the number is equivalent to the number of queries in Table 2 (as those methods deploy one-click oracles). Tables 2b , 2d, and 2f give the average and maximum click numbers of a workload for `classical`, in comparison to the corresponding numbers of `FIO`. The difference between the two methods widened continuously as k increased, with `FIO` achieving a "click saving" of more than 50% compared to `classical` at $k = 10$.

The cost of IGS depends heavily on the level of the target vertex. To demonstrate the effect of level, we zoomed into the results in Table 2 for $k = 5$. For each workload, we grouped the targets therein by level and measured, for each algorithm, the average number of queries/clicks required to find the targets of each specific level. Figure 4a (resp., 4b) presents the query (resp., click) number as a function of level for `Amazon`. The results for `ImageNet` and `WordNet` are shown in Figures 4c-f (note: these two DAGs have no leaves at level 1). `FIO` and `classical` needed significantly fewer queries than the two taciturn algorithms in finding deep targets. Furthermore, as the level increases, the superiority of `FIO` over `classical` in minimizing clicks became increasingly evident.

Results on Synthetic Data. We now eliminate (i) `classical` because it is dominated by `FIO` (in both query and click numbers) and (ii) `LMNT-taciturn` due to its dominance by `new-taciturn`. Our next experiments aim to study `FIO` and `new-taciturn` — the two new algorithms spawned from our framework — as the input DAG changes in a certain aspect. Synthetic data fit the purpose very well. Each experiment below is characterized by parameters n, d, r, and k, where the first three parameters indicate the synthetic DAG used (see Section 5.1) and k, as before, is the size of the query set Q given to the oracle each time.

To start with, we set n, d, and r to their default values and, for each $k \in [1, 10]$, used `FIO` and `new-taciturn` to process a random workload. Figure 5a plots each method's average number of queries (per instance in the workload) as a function of k. Note that each number can be alternatively interpreted as the average number of clicks (the oracles of `FIO` and `new-taciturn` are one-click). Next, we set k to the median value 5 and repeated the above experiment by varying one of the parameters n, d and r. Figures 5b, 5c, and 5d plot the number of queries as a function of n, d, and r, respectively.

Each method's behavior in Figures 5a-c closely aligns with its theory-predicted cost: $O(\log_k n + (d/k) \log_d n)$ for `FIO` and $O(\log n + (d/k) \log_d n)$ for `new-taciturn`. Figure 5d, on the other hand, reveals the interesting phenomenon that both methods incurred lower cost as the DAG deviated further away from a tree. Recall that a higher cross ratio r induces more cross edges, which intuitively strengthens a node's capability to reach the target. As a result, both algorithms were able to locate the target with less queries.

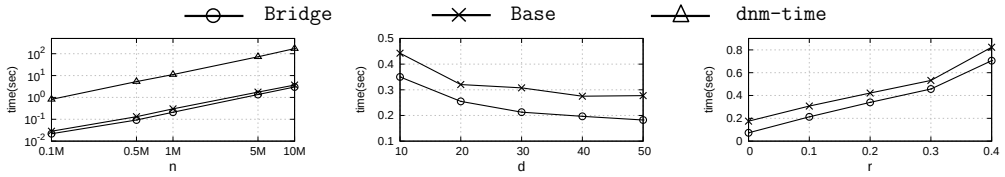
5.3 Evaluation of Computation Efficiency

We now proceed to evaluate the benefits brought by our new algorithms for computing the HPDFS-tree. All experiments in this subsection were conducted on a machine with an Intel CPU running at 3.6GHz and 8 GB of memory; the operating system was Ubuntu 22.04. Our evaluation inspected three HPDFS-tree algorithms:

dataset	dnm-time HPDFS (IGS)	Base HPDFS (IGS)	Bridge HPDFS (IGS)
Amazon	89 (111)	6.1 (7.6)	2.5 (3.6)
ImageNet	130 (190)	7.8 (11)	3.4 (5.5)
WordNet	515 (759)	28 (59)	12 (24)

unit: millisecond

Table 3. HPDFS and IGS computation time (real data)



(a) Vs. n ($d = 30, r = 0.1$) (b) Vs. d ($n = 1$M, $r = 0.1$) (c) Vs. r ($d = 30, n = 1$M)

Fig. 6. HPDFS computation time (synthetic data)

- `dnm-time`: The $O(dnm)$-time algorithm of [22], as reviewed in Section 2. The implementation was exactly the one in [16].
- `Base`: The $O(nm)$-time algorithm described in Section 4.1.
- `Bridge`: The $O(m + n \cdot \Delta)$-time algorithm in Theorem 5.

Results on real data. Our first set of experiments in this subsection utilized each competing method to (i) build the HPDFS-tree of each real dataset, and (ii) execute a full workload (defined in Section 5.2) under the FIO-oracle with $k = 5$. We measured:

- the CPU time to compute the HPDFS-tree on the input G;
- the average CPU time of an IGS instance in the workload (each instance invokes a recursion, where at each recursion level the HPDFS-tree of a subgraph of G is computed).

The results are in Table 3, with IGS time shown in brackets.

On all datasets, the two proposed algorithms were able to produce the HPDFS-tree significantly faster than the existing `dnm-time` method, achieving a speedup of over an order of magnitude in all scenarios. Such a vast efficiency gain was translated to the total IGS time as well: the overall CPU duration of an IGS instance was also shortened by more than an order of magnitude. Between our own algorithms, `Bridge` was roughly 50% faster in both HPDFS-tree construction and overall IGS computation.

Results on synthetic data. We continued with a similar set of experiments on synthetic datasets, where we adjusted one of the parameters n, d, and r while keeping the others fixed. Figure 6a (resp., 6b and 6c) plots the HPDFS construction time as a function of n (resp., d and r). The time escalation with n in Figure 6a is consistent with these algorithms' runtime complexities. When n reached 10 million, `dnm-time` took more than 170 seconds, while our algorithms finished within 5 seconds.

In Figure 6b, the time decreased as d became larger, which can be attributed to the fact that a larger d led to a shallower HPDFS-tree. The result of Figure 6c implied that `Bridge` gradually lost its superiority over `Base` when the input DAG G departed increasingly away from a tree. Note that `dmn-time` was removed from Figures 6b and 6c as the method was not competitive at all.

5.4 IGS in WordNet Categorization

This subsection demonstrates the usefulness of IGS in a concrete application: word categorization in WordNet. Given the widespread success of LLMs, our study was designed to address two timely

	G2V-1	G2V-10	G2V-100	LLM-Topdown	LLM-IGS	L-H-IGS	L-H-IGS$^+$	H-IGS
number of successes (out of 30 synsets)	0	14	22	0	0	21	26	30
average number of queries per synset	-	-	-	-	7	11	12	13
average number of hypernym tests by human per synset	1	10	100	0	0	32	37	43

Table 4. Aggregate results of synset categorization in WordNet

questions. First, is IGS subsumed by LLMs in functionality in this context? Second, can LLMs be leveraged to assist humans in IGS?

As explained in Section 5.1, WordNet is a DAG $G = (V, E)$ where nodes are synsets and a (directed) edge (u, v) indicates synset u being a hypernym of synset v. Our setup simulated the scenario where linguists needed to create new leaf synsets. For the experiments to be meaningful, it is imperative to have non-disputable ground truths; and we achieved the purpose as follows. Let z be a leaf synset in WordNet with a single parent t (that is, an in-neighbor of z). We first removed z from WordNet — denote the resulting DAG as $G_{\setminus z}$ — and then tried to put it back. As the ground truth, a method should locate t as the "target" in $G_{\setminus z}$. We say that the method *succeeds* if it manages to do so, or *fails* otherwise.

We examined several methods that represented different degrees of integration between IGS and an LLM (we used OpenAI's GPT-4o-2024-08-06 model, abbreviated as *the LLM* henceforth).

- Gloss2Vec-K: In WordNet, every synset u comes with a gloss (a textual explanation), which we denote as $gloss(u)$. OpenAI offered a utility (text-embedding-3-small) for converting $gloss(u)$ to a vector $vec(u)$. The conversion has the property that if $gloss(u)$ is semantically close to $gloss(v)$, then $vec(u)$ and $vec(v)$ tend to be close in cosine distance. In Gloss2Vec-K, we first obtained the K synsets in $V \setminus \{z\}$ whose vectors had the smallest cosine distances to $vec(z)$; call those synsets the K *nearest neighbors* (NN) of z. A human then examined the K NNs to identify the most specific hypernym of z. If t (the original parent of z) was one of the K NNs, the method succeeded; otherwise, it failed.
- LLM-Topdown: This method used the LLM to find the target t from $G_{\setminus z}$ in a "one-level-at-a-time" manner. Specifically, at each non-leaf node u (initially, u was the root), we fed the LLM with $gloss(z)$ and the glosses of all the children of u. The LLM then picked the child v of u most likely to be a hypernym of z, after which the process was repeated at v. The LLM returned u if it considered that no such v existed.
- LLM-IGS: This method applied our FIO algorithm (Section 5.2) on $G_{\setminus z}$ by having the LLM play the oracle. Specifically, given a query $\bar{Q}$, we fed the LLM with $gloss(z)$ and the glosses of all the nodes in $\bar{Q}$. Then, the LLM returned what it considered was the leftmost hypernym of z (in the sequence $\bar{Q}$), or indicated the absence of any hypernym in $\bar{Q}$.
- LLM-Human-IGS: This method implemented FIO on $G_{\setminus z}$ by involving both humans and the LLM. Given a query $\bar{Q}$, we fed the LLM with the same information as in LLM-IGS. However, here, the LLM was asked to remove from $\bar{Q}$ the nodes that were not hypernyms of z. The human then inspected the remaining nodes of $\bar{Q}$ to identify the letmost hypernym of z. If none was found, the human was then asked to inspect the nodes removed by the LLM to either find the leftmost hypernym of z in $\bar{Q}$ or confirm that none existed.
- LLM-Human-IGS$^+$: Same as LLM-Human-IGS except that the human handled $\bar{Q}$ directly when the root of $G_{\setminus z}$ was in $\bar{Q}$.
- Human-IGS: The FIO algorithm with a human as the oracle.

Human answers were always correct (in reality, linguists' opinions prevail). All the FIO implementations used the parameter $k = 5$ (as in Figure 4). Our experiments used 30 synsets selected in

word	G2V-10	G2V-100	L-H-IGS	L-H-IGS$^+$	H-IGS
evergreen	✗	✓	✓, 7, 24	✓, 7, 24	✓, 7, 26
wassail	✓	✓	✓, 11, 19	✓, 11, 19	✓, 11, 23
allusion	✓	✓	✗, 8, 23	✗, 8, 25	✓, 10, 26
cavalier	✗	✓	✓, 9, 12	✓, 9, 14	✓, 9, 16
sublimate	✗	✗	✓, 9, 28	✓, 9, 29	✓, 9, 29
eddy	✗	✗	✗, 2, 3	✗, 8, 20	✓, 11, 22
snivel	✗	✓	✓, 9, 16	✓, 9, 17	✓, 9, 24
grandiloquence	✗	✗	✓, 12, 35	✓, 12, 37	✓, 12, 46
bevy	✓	✓	✗, 7, 16	✓, 9, 16	✓, 9, 20
psephology	✗	✓	✓, 12, 18	✓, 12, 20	✓, 12, 31
hatching	✓	✓	✓, 9, 14	✓, 9, 14	✓, 9, 18
disbursement	✗	✓	✗, 2, 3	✓, 12, 32	✓, 12, 34
euphoria	✓	✓	✓, 10, 17	✓, 10, 20	✓, 10, 24
garnish	✓	✓	✓, 14, 47	✓, 14, 48	✓, 14, 50
defenestration	✓	✓	✗, 2, 3	✓, 9, 26	✓, 9, 26
scapegoat	✗	✗	✓, 11, 31	✓, 11, 32	✓, 11, 35
adversary	✗	✗	✓, 82, 401	✓, 82, 401	✓, 82, 401
gossamer	✗	✓	✓, 10, 23	✓, 10, 24	✓, 10, 25
tribulation	✗	✓	✗, 2, 3	✓, 10, 20	✓, 10, 24
paraphernalia	✓	✓	✓, 10, 27	✓, 10, 29	✓, 10, 31
winnow	✓	✓	✗, 2, 3	✓, 11, 31	✓, 11, 31
succor	✓	✓	✓, 10, 29	✓, 10, 31	✓, 10, 35
ethos	✗	✗	✓, 8, 23	✓, 8, 25	✓, 8, 28
memento	✓	✓	✓, 11, 22	✓, 11, 23	✓, 11, 27
skirl	✗	✗	✗, 2, 3	✗, 8, 20	✓, 20, 75
Byzantine	✗	✓	✗, 11, 38	✗, 11, 37	✓, 16, 62
propinquity	✓	✓	✓, 11, 20	✓, 11, 21	✓, 11, 25
jeremiad	✓	✓	✓, 9, 20	✓, 9, 20	✓, 9, 25
dearth	✓	✓	✓, 10, 20	✓, 10, 21	✓, 10, 26
tchotchke	✗	✗	✓, 6, 11	✓, 6, 12	✓, 6, 20

Table 5. Detailed results of synset categorization in WordNet

a non-subjective manner. Specifically, we took the last 30 nouns on Merrian-Webster's "word of the day"[4] before 31 Dec 2024 satisfying the condition that the noun appeared in a unique synset z of WordNet. The synset z was then selected for our tests. All these nouns are listed in the first column of Table 5.

For each selected synset z, we tested all the methods mentioned earlier (for Gloss2Vec-K, we evaluated $K = 1$, 10, and 100). For each method, we measured the number of synsets on which it succeeded. In addition, for every method running on FIO, we measured the number of queries issued on each word. Finally, for all methods, we also recorded the number of hypernym tests that the human linguist had to perform, i.e., how many times the human was summoned to manually decide whether a synset was a hypernym of z. This number represents the amount of "mental effort" by the human and is a metric applicable to all the methods (the number of queries is inapplicable to Gloss2Vec-K). Recall that Gloss2Vec-K requires a human to perform K hypernym tests per synset.

Table 4 presents our test results at the aggregate level; the numbers at the last two rows are averages over the 30 selected synsets. Table 5 gives the detailed results on every synset of all methods except Gloss2Vec-1, LLM-Topdown, and LLM-IGS, which are omitted because they failed on every

[4]See www.merriam-webster.com/word-of-the-day.

synset. Here, a ✓ (resp., ✗) symbol indicates success (resp., failure). Each result of LLM-Human-IGS, LLM-Human-IGS$^+$, and Human-IGS is presented in the format of "✓/✗, x, y", where x is the number of queries issued, and y is the number of hypernym tests by the human. The value x is also the number of mouse clicks (because FIO is a one-click oracle).

It is worth noting from Table 5 that the synset of "adversary" was an outlier. The target t — namely the synset's original parent in WordNet — was "person", which has an unusually large number of child nodes (the number is 399). To confirm the target, a human must check all those children (to make sure they are not hypernyms of "adversary"), which explains the high costs of LLM-Human-IGS, LLM-Human-IGS$^+$, and Human-IGS (note: the number 401 is roughly 5 times of 82 because $k = 5$). If this outlier case was discounted, then LLM-Human-IGS, LLM-Human-IGS$^+$, Human-IGS required the human to perform only 19, 24, and 30 hypernym tests on average per synset, respectively.

In practice, the purpose of labeling (as is done in synset categorization) is for training new models. As training is extremely expensive in terms of both the electricity consumed and running time, companies today launch a massive training process only after they have secured high-quality labeled data. This is especially true considering that labeling is actually significantly cheaper in comparison. Our experiment results strongly suggest that garnering reliable human inputs is still the best approach to ensure the quality of labeling, which reaffirms the motivation of IGS.

The LLM, if run with no human intervention, did not produce accurate categorization at all. However, it could be used to reduce human effort if a moderate drop in accuracy is deemed acceptable. LLM-Human-IGS$^+$ appears to offer a reasonable tradeoff: it saved about 20% of humans' efforts compared to Human-IGS, but erred on 4 synsets (out of 30). In any case, the most serious weakness of LLMs in general is that they offer no guarantees. They can perform terribly in certain situations, and it is exceedingly difficult (if not impossible) to predict what those situations are.

As shown in Table 5, Human-IGS usually required a human to perform between 20 and 30 hypernym tests for a synset. Considering that the entire WordNet has over 82k synsets, we believe that this method makes an appealing solution in practice.

6 Related Work

The history of IGS, strictly speaking, could be traced back to the last century, although early work focused on the special scenario where the input G is a tree — rather than a DAG — and the value of k is fixed to 1 (in which case the classical, taciturn, and FIA oracles are all the same). Specifically, Ben-Asher and Farchi [1] studied the problem in as early as 1997, but they did not manage to solve the problem optimally. The first optimal algorithm on (trees and $k = 1$) was developed by Laber and Nogueira [12] in 2001. Interestingly, since then, their results have been re-discovered at least twice: in 2006 [8] and 2016 [9], respectively.

Research on DAGs did not commence until 2011, when Parameswaran et al. [19] conceived the concept of "human-assisted graph search". The concept is similar to IGS except that no interaction is permitted. Instead, all questions to the oracle must be prepared "in one go". The objective is to get, as much as possible, close to the target vertex (rather than finding the target precisely) subject to how many questions can be prepared in advance. However, in many scenarios, the number of questions must be huge to guarantee locating a concept (a.k.a. a vertex in G) that is "not too distant" from the target t.

The term IGS (interactive graph search) was coined by Tao et al. [22] in 2019. Departing from the all-in-one-go approach of [19], they introduced interaction between an algorithm and a human annotator, which not only enables the algorithm to precisely pinpoint the target t but also significantly reduces the number of required questions compared to [19]. They were the first to define

the HPDFS procedure and showed how to utilize the procedure to design an IGS algorithm for the classical oracle with non-trivial guarantees.

Improving the result of [22], Lu et al. [15] in 2022 optimally settled IGS for the classical oracle. They achieved the purpose by establishing and harnessing a set of interesting graph-theoretic properties of HPDFS. In [16], which is the long version of [15], Lu et al. formulated the taciturn oracle to address the issue with the classical oracle's user-unfriendliness. However, as mentioned in Section 1.1, their taciturn IGS algorithm is nearly-optimal, leaving a gap between the upper bound and the lower bound. We have managed to close the gap in this work.

Since [22], several other variants of IGS have been studied in the literature. Assuming G to be a tree, Li et al. [14] considered the "multiple-choice oracle" that, when given a vertex u in G, returns either the child of u (in the tree G) being an ancestor of the target t, or a token indicating the absence of such a child. Again assuming G to be a tree, Zhu et al. [26] explored an IGS situation where multiple target vertices are present. In [4], Cong et al. investigated how to minimize the expected cost of IGS with $k = 1$ when a distribution of the target vertex is available (see [6, 7, 11, 13, 17, 18] for earlier work that dealt with the same setup but under the constraint that G is a tree and the distribution is uniform). Assuming G to be a tree, Cong et al. [3] examined a noisy version of IGS where the oracle may return incorrect answers. In [25], Zhao et al. proposed a method to accelerate IGS for image categorization, which utilizes similarity search to shrink the concept (DAG) hierarchy G. They also gave an IGS algorithm whose cost is lower than that of [22] (but they did not seem aware of the results in [15, 16]).

Finally, we note that machine learning methods have recently been deployed to deal with taxonomy expansion scenarios similar to our setup in Section 5.4. The interested readers may see [20, 21, 24] and the references therein.

7 Conclusions

IGS, which aims to find a hidden vertex in a DAG with the least interaction rounds between an algorithm and an oracle, has established its importance in diverse applications. However, the existing algorithms suffer from at least one of the following drawbacks. First, they may require more interaction than what is predicted to be necessary by theory. Second, they may demand considerable monotonous inputs from the (human) oracle. In addition, previous research has overlooked the issue of scalable computation of the HPDFS tree even though this is a crucial component in the state-of-the-art solutions. The current paper has presented a systematic study to address all these problems. We have developed an algorithmic framework that permits a designer to devise robust IGS algorithms by filling in the implementation of two simple black-box operations. The framework yields concrete algorithms under various oracles that achieve asymptotically the minimum interaction with the least human inputs. In addition, we accompany our framework with a new method for computing the HPDFS tree with a significantly reduced time complexity. Finally, we have presented an extensive experimental evaluation confirming that the proposed solutions outperformed their competitors in all the scenarios examined.

Acknowledgments

Yufei Tao's research was supported in part by GRF projects 14222822 and 14203421 from HKRGC. We also thank the anonymous reviewers for their constructive suggestions on an earlier version of the paper.

References

[1] Yosi Ben-Asher and Eitan Farchi. 1997. *The cost of searching in general trees versus complete binary trees.* Technical Report.

[2] Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. 1999. Optimal Search in Trees. *SIAM Journal of Computing* 28, 6 (1999), 2090–2102.
[3] Qianhao Cong, Jing Tang, Kai Han, Yuming Huang, Lei Chen, and Yeow Meng Chee. 2022. Noisy Interactive Graph Search. In *Proceedings of ACM Knowledge Discovery and Data Mining (SIGKDD)*. 231–240.
[4] Qianhao Cong, Jing Tang, Yuming Huang, Lei Chen, and Yeow Meng Chee. 2022. Cost-Effective Algorithms for Average-Case Interactive Graph Search. In *Proceedings of International Conference on Data Engineering (ICDE)*. 1152–1165.
[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. The MIT Press.
[6] Pilar de la Torre, Raymond Greenlaw, and Alejandro A. Schäffer. 1995. Optimal Edge Ranking of Trees in Polynomial Time. *Algorithmica* 13, 6 (1995), 592–618.
[7] Dariusz Dereniowski. 2008. Edge ranking and searching in partial orders. *Discrete Applied Mathematics* 156, 13 (2008), 2493–2500.
[8] Dariusz Dereniowski and Marek Kubale. 2006. Efficient Parallel Query Processing by Graph Ranking. *Fundamenta Informaticae* 69, 3 (2006), 273–285.
[9] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. 2016. Deterministic and probabilistic binary search in graphs. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*. 519–532.
[10] Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Kumar Saini, Saurabh Bagchi, and Murat Kocaoglu. 2022. Root Cause Analysis of Failures in Microservices through Causal Discovery. In *Proceedings of Neural Information Processing Systems (NIPS)*.
[11] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. 1991. On an edge ranking problem of trees and graphs. *Discrete Applied Mathematics* 30, 1 (1991), 43–52.
[12] Eduardo Sany Laber and Loana Tito Nogueira. 2001. Fast Searching in Trees. *Electronic Notes in Discrete Mathematics* 7 (2001), 90–93.
[13] Tak Wah Lam and Fung Ling Yue. 2001. Optimal Edge Ranking of Trees in Linear Time. *Algorithmica* 30, 1 (2001), 12–33.
[14] Yuanbing Li, Xian Wu, Yifei Jin, Jian Li, and Guoliang Li. 2020. Efficient Algorithms for Crowd-Aided Categorization. *Proceedings of the VLDB Endowment (PVLDB)* 13, 8 (2020), 1221–1233.
[15] Shangqi Lu, Wim Martens, Matthias Niewerth, and Yufei Tao. 2022. Optimal Algorithms for Multiway Search on Partial Orders. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*. 175–187.
[16] Shangqi Lu, Wim Martens, Matthias Niewerth, and Yufei Tao. 2023. Partial Order Multiway Search. *ACM Transactions on Database Systems (TODS)* 48, 4 (2023), 10:1–10:31.
[17] Shay Mozes, Krzysztof Onak, and Oren Weimann. 2008. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1096–1105.
[18] Krzysztof Onak and Pawel Parys. 2006. Generalization of Binary Search: Searching in Trees and Forest-Like Partial Orders. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 379–388.
[19] 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. *Proceedings of the VLDB Endowment (PVLDB)* 4, 5 (2011), 267–278.
[20] Jiaming Shen, Zhihong Shen, Chenyan Xiong, Chi Wang, Kuansan Wang, and Jiawei Han. 2020. TaxoExpan: Self-supervised Taxonomy Expansion with Position-Enhanced Graph Neural Network. In *Proceedings of International World Wide Web Conferences (WWW)*. 486–497.
[21] Yanzhen Shen, Yu Zhang, Yunyi Zhang, and Jiawei Han. 2024. A Unified Taxonomy-Guided Instruction Tuning Framework for Entity Set Expansion and Taxonomy Expansion. *arXiv preprint arXiv:2402.13405* (2024).
[22] Yufei Tao, Yuanbing Li, and Guoliang Li. 2019. Interactive Graph Search. In *Proceedings of ACM Management of Data (SIGMOD)*. 1393–1410.
[23] Robert Endre Tarjan. 1974. A Note on Finding the Bridges of a Graph. *Information Processing Letters (IPL)* 2, 6 (1974), 160–161.
[24] Yue Yu, Yinghao Li, Jiaming Shen, Hao Feng, Jimeng Sun, and Chao Zhang. 2020. STEAM: Self-Supervised Taxonomy Expansion with Mini-Paths. In *Proceedings of ACM Knowledge Discovery and Data Mining (SIGKDD)*. 1026–1035.
[25] Zhuowei Zhao, Junhao Gan, Jianzhong Qi, and Zhifeng Bao. 2024. Efficient Example-Guided Interactive Graph Search. In *Proceedings of International Conference on Data Engineering (ICDE)*. 342–354.
[26] Xuliang Zhu, Xin Huang, Byron Choi, Jiaxin Jiang, Zhaonian Zou, and Jianliang Xu. 2021. Budget Constrained Interactive Search for Multiple Targets. *Proceedings of the VLDB Endowment (PVLDB)* 14, 6 (2021), 890–902.

Received October 2024; revised January 2025; accepted February 2025