Parameterizing the Complexity of Finding Long Paths in DAGs

Ronak Bhadra
Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, India

Saurya Singh
Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, India

Raghunath Tewari
Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, India

Abstract

Given a graph G and two vertices s and t, the Long Path problem asks whether there exists a path of length at least k from s to t. For general graphs, this problem is NP-hard when k is part of the input. For directed acyclic graphs (DAGs), however, it is solvable in polynomial time and is NL-complete.

A nondeterministic logspace computation is said to be *unambiguous* if, on every input, there is at most one accepting computation path. The class UL consists of all problems solvable by such machines, and whether NL = UL is an open question.

In this work, we study the unambiguous complexity of the Long Path problem on DAGs under parameterization. Specifically, we consider the problem of deciding whether there exists a path of length at least $n - k$ between two given vertices in a DAG. Bhadra and Tewari [2] showed that this problem can be solved in unambiguous and co-unambiguous $O(k \log n)$ space. We improve this result by giving an algorithm that runs in unambiguous $O(k + \log n)$ space. Additionally, we obtain an algorithm that achieves unambiguous and co-unambiguous $O(k \log n)$ space while running in time polynomial in both n and k, improving the previous $O^*(n^k)$ time bound.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Unambiguous Computations, Directed Acyclic Graphs, Space Complexity

Digital Object Identifier 10.4230/LIPIcs.MFCS.2026.73

Funding This work has been partly funded by Research-I foundation.

1 Introduction

Given a graph G and two vertices s and t, deciding whether there exists a path of length at least k from s to t is known as the *Long Path* problem. The Long Path problem is NP-hard for general graphs, since the Hamiltonian Path problem – known to be NP-hard – is a special case corresponding to $k = n$, where n is the number of vertices in the graph. However, for directed acyclic graphs (DAGs), the Long Path problem is solvable in linear time and also in nondeterministic logspace (NL). Moreover, it is NL-hard even for DAGs, since s-t reachability corresponds to the case $k = 1$. Thus, the problem is para-NL-hard with respect to the parameter k.

A nondeterministic logspace computation is said to be *unambiguous* if, on every input, there is at most one accepting computation path. The class UL consists of all decision problems that can be solved by unambiguous logspace machines, and the class coUL contains all problems whose complements are in UL. Reinhardt and Allender in their seminal work [10] introduced the technique of double-inductive counting, based on the inductive counting method of Immerman-Szelepcsényi [7, 12], which they used to show that UL/*poly* = NL/*poly*. Improving upon the techniques of Reinhardt and Allender, Limaye, Mahajan, and Nimbhorkar

showed that the length of the longest path from the source to any vertex in a single-source max-unique DAG can be decided in $\mathsf{UL} \cap \mathsf{coUL}$ [8]. Meanwhile, Pavan, Tewari and Vinodchandran showed that reachability in graphs having at most K paths can be decided in unambiguous space $O(\log K + \log n)$ [9]. We call a graph max-K when there are at most K longest paths between any two vertices in the graph and source-rooted max-K when there are at most K longest paths from any source to any vertex in the graph. Extending the techniques of Limaye et. al.[8] and Pavan et. al.[9], Dhayal, Sarma, and Sawlani showed that reachability and the Long Path problem in max-K DAGs can be decided in unambiguous and co-unambiguous $O(\log K + \log n)$ space, provided a topological ordering of the vertices is given [4].

Recently, Bhadra and Tewari considered an inverted parameterization of the Long Path problem [2], where the task is to decide whether there exists a path of length at least $n - k$ between two vertices. They showed that this problem can be solved in unambiguous and co-unambiguous space $O(k \log n)$, although the running time of their algorithm is $O^*(n^k)$. This parameterization can be viewed as the Long Path analogue of the Short Path problem, which asks whether there exists a path of length at most k between two vertices. While the Short Path parameterization concerns detecting very short paths efficiently, this version of Long Path asks whether very long paths can be detected efficiently. The best known parameterized space bound for the Short Path problem is $O(\log k \log n)$ via Savitch's Theorem [11]. To the best of our knowledge, no para-L or para-UL bound (as defined by Flum and Grohe [6]) is known for Short Path. In fact, the Short Path problem, also known as Distance, has been shown to be complete for the class para-βL[5].

We show that deciding whether there exists a path of length at least $n - k$ from s to t in a DAG G can be done in unambiguous $O(k + \log n)$ space; however, our algorithm does not guarantee the same bound for co-unambiguous space. Thus, the problem lies in para-UL with respect to the parameter k.

Using ideas from [4], we give an unambiguous $O(\log K + \log n)$ space algorithm to recognize max-K DAGs and an unambiguous and co-unambiguous $O(\log K + \log n)$ space algorithm to decide reachability in max-K DAGs. Finally, we show that if a DAG contains a path of length at least $n - k$, then it must be max-2^k. Combining this structural bound with the above algorithms yields our unambiguous $O(k + \log n)$ space algorithm. Moreover, our algorithm rejects any input that is not a DAG, thereby removing the need for a promise that the input graph is acyclic, which is otherwise NL-hard to check independently.

We note that Dhayal, Sarma, and Sawlani [4] gives an algorithm that can be used to recognize single-source max-K DAGs in unambiguous $O(\log K + \log n)$ space, even if the original purpose of the algorithm was not intended to be this. They also give an algorithm to decide reachability in max-K DAGs in unambiguous and co-unambiguous $O(\log K + \log n)$ space if a topological ordering of the DAG is provided, which is NL-hard to derive for general DAGs [13]. Our main contribution here is the following. First, we generalize the recognition result from single-source DAGs to all DAGs. Second, we extend the reachability algorithm to all max-K DAGs, removing the requirement that a topological ordering be provided as part of the input. Third, we make both problems promise-independent by removing the assumption that the input belongs to a particular promise class (such as DAGs or max-K DAGs), although some nuances remain, which we will discuss now.

We observe that the algorithm of Dhayal, Sarma, and Sawlani [4] to recognize single-source max-K DAGs does not yield an unambiguous algorithm for the complement problem. As a result, our algorithms for recognizing max-K DAGs and for deciding the Long Path problem also achieve unambiguous space bounds of $O(\log K + \log n)$ and $O(k + \log n)$, respectively, but corresponding co-unambiguous bounds are not guaranteed. Similarly, reachability in

max-K DAGs can be decided in unambiguous and co-unambiguous $O(\log K + \log n)$ space under the promise that the input is a max-K DAG, but without this promise, only the unambiguous bound is guaranteed.

Eric Allender notes in [1] that "we have no good candidates for being in UL and not in coUL." Problems such as recognizing max-unique DAGs or deciding whether there exists a path of length at least $n - O(\log n)$ in a DAG may serve as potential candidates. A separation between UL and coUL would have major consequences in complexity theory, as it would imply $\mathsf{L} \neq \mathsf{NL}$ and hence $\mathsf{L} \neq \mathsf{P}$. On the other hand, it is widely believed that $\mathsf{UL} = \mathsf{coUL}$, in which case problems such as recognizing max-unique DAGs would need to be shown to lie in coUL. Thus, studying these problems may provide further insight into the relationship between these classes.

Finally, we improve the running time of the algorithm of Bhadra and Tewari from $O^*(n^k)$ to polynomial in n, while maintaining the unambiguous and co-unambiguous space bound of $O(k \log n)$. In line with how the class XNLP is defined [3, 5], we define the class $\mathsf{X}(\mathsf{UL} \cap \mathsf{coUL})\mathsf{P}$ as the the class of parameterized problems such that an instance of size n with parameter k can be solved by an unambiguous and co-unambiguous non-deterministic Turing Machine in time $f(k)\text{poly}(n)$ and space $f(k)\log(n)$. Our methods prove that the problem of deciding if there exists a path of length at least $n - k$ between two given vertices in a DAG is in $\mathsf{X}(\mathsf{UL} \cap \mathsf{coUL})\mathsf{P}$ when parameterized by k.

2 Recognizing Single-Source DAGs that are max-K with respect to the Source

In this section, we will show that single-source DAGs where there are at most K longest paths from the source to all other vertices (in other words, max-K with respect to the source) can be recognized in unambiguous $O(\log K + \log n)$ space. Also, the length of the longest path from the source to any other vertex can also be decided in unambiguous $O(\log K + \log n)$ space for such graphs.

Given a single-source DAG G and a vertex v as input, the nondeterministic algorithm Main-Max-Poly in [4] operates using $O(\log K + \log n)$ space and behaves as follows:

- If there are at most K longest paths from the source to all other vertices, then the algorithm returns the length of the longest path from the source of G to v along a unique computation path, while all other computation paths reject.
- If there are more than K longest paths from the source to any other vertex, then the algorithm rejects along all computation paths.

This algorithm can be further modified to reject along all computation paths if the input graph is not a DAG. To this end, we add the following line at the end of the subroutine Find-Fault-Max (after line 35) in [4]:

if $c_k \neq n$ then REJECT.

A vertex u is counted in c_k while updating c_k from c_{k-1} in the subroutine Update-Max-Poly only if the length of the longest walk from the source to each in-neighbor of u is strictly less than k. Any vertex that lies on a directed cycle admits arbitrarily long walks from the source (by traversing the cycle repeatedly), and therefore cannot satisfy this condition. Hence, no vertex that is part of a directed cycle is ever counted in c_k for any value of k. Consequently, if the input graph contains a directed cycle, the value of c_k in the subroutine Find-Fault-Max can never become equal to n, and the added check correctly causes rejection.

It follows that single-source DAGs having at most K longest paths from the source to any vertex can be recognized in unambiguous $O(\log K + \log n)$ space, even without assuming that the input graph is a DAG.

Thus, we have the following theorems.

▶ **Theorem 1.** *The set of single-source DAGs max-K with respect to the source can be recognized in unambiguous $O(\log K + \log n)$ space.*

▶ **Theorem 2.** *Given a single-source DAG G max-K with respect to the source, the length of the longest path from the source to any vertex v in G can be found out in unambiguous $O(\log K + \log n)$ space.*

We note, however, that the above algorithm does not yield the same complexity bound for the complement of the problem. This is in contrast to the case of recognizing graphs having at most K shortest paths from a vertex v to any other vertex, where both the problem and its complement can be decided in unambiguous $O(\log K + \log n)$ space. The source of this asymmetry lies in how nondeterminism is used to certify upper bounds on shortest versus longest walks.

For simplicity, consider the case $K = 1$, corresponding to min-unique graphs and single-source DAGs max-unique with respect to source. To decide whether the shortest walk to a vertex v has length at least i, it suffices to nondeterministically guess a walk of length less than i to certify that the shortest walk to a vertex v has length less than i. By the double inductive counting method [10], vertices whose shortest walks are of length less than i are guaranteed to have unique shortest paths, ensuring that only a single computation path succeeds to find the shortest path.

In contrast, to decide whether the longest walk to a vertex v has length at least i, it is not enough to guess a walk of length less than i, since the existence of such a walk does not preclude the existence of a longer one. As introduced in [8] and used in [4], the algorithm therefore also nondeterministically guesses a longest path of length at least i before declaring that the longest path up to the vertex is of length at least i. However, while the modified double inductive counting method guarantees uniqueness of longest paths only for vertices whose longest walks have length less than i, it does not ensure uniqueness for vertices whose longest walks have length at least i. As a result, if the input graph is not max-unique, the nondeterministic algorithm may accept the existence of a longest walk of length at least i along multiple computation paths. Although all computation paths eventually reject when the graph is not max-unique, there need not be a unique rejecting computation path witnessing non-max-uniqueness. Hence, the complement of the problem of recognizing single-source DAGs max-unique with respect to the source is not trivially in UL, though the problem is in UL.

3 Deciding Reachability in Single-Source Max-K DAGs

In this section, we will show that given a single-source max-K DAG G, reachability between any two vertices in G can be decided in unambiguous $O(\log K + \log n)$ space.

We get the following lemma from [4] (Lemma 4.1).

▶ **Lemma 3.** *Given a topological ordering of a DAG G, reachability in G can be log-space many-one reduced to finding long path from the source to some vertex in a single-source DAG G', such that G' is max-K if G is max-K.*

It is in general NL-hard to find the topological ordering of a given DAG [13]. However, given a single-source max-K DAG G, a topological ordering of G can be derived in unambiguous $O(\log K + \log n)$ space. We can do it as follows. According to Theorem 2, we can find the length of the longest path from the source to any vertex v in G in unambiguous $O(\log K + \log n)$ space. In order to figure out the index of a vertex v of G in the topological ordering of G, we find the length of the longest path from the source of G to v (say i). We then find out how many vertices u (say m) are there in G such that the length of the longest path from the source of G to u is either less than i, or it is equal to i and the vertex label of u is less than the vertex label of v. We assign m to be the index of v in the topological ordering. If the length of the longest path from the source to a vertex x is larger than the length of the longest path from the source to another vertex y, there cannot be an edge from y to x in G since that would imply there is a longer path to y from the source than the longest path to x from the source, that can be formed by taking the longest path to x from the source and extending it up to y by adding the edge from y to x. Our scheme for topological ordering correctly gives x a smaller index than y in this case. The whole computation happens in unambiguous $O(\log K + \log n)$ space, because each length of longest path query is computed successfully along exactly one computation path using a non-deterministic $O(\log K + \log n)$ space Turing machine.

This combined with Lemma 3 shows that given a single-source max-K DAG G, reachability in G can be log-space many-one reduced to finding long path from the source to some vertex in a single-source DAG G', such that G' is max-K. Now, by Theorem 2, we can compute the length of the longest path from the source to some vertex in G' in unambiguous $O(\log K + \log n)$ space. This implies that reachability in G can be decided in unambiguous and co-unambiguous $O(\log K + \log n)$ space.

Therefore, we get the following theorem.

▶ **Theorem 4.** *Given a single-source max-K DAG G, reachability in G can be decided in unambiguous and co-unambiguous $O(\log K + \log n)$ space.*

4 Reducing Max-K-ness of DAGs to Max-K-ness of Single-Source DAGs

In this section, we show that the problem of deciding whether a DAG is max-K can be logspace reduced to the problem of deciding whether a *single-source* DAG is max-K. For this, we give a logspace construction of a single-source DAG from a given DAG such that max-K-ness is preserved. The construction also preserves reachability, implying that reachability in max-K DAGs is logspace reducible to reachability in single-source max-K DAGs. We state this in the following theorem.

▶ **Theorem 5.** *Given a DAG G, we can construct in logspace a single-source DAG G' containing all the vertices in G along with other additional vertices such that*

- *G' is max-K if and only if G is max-K.*
- *A vertex v is reachable from vertex u in G if and only if vertex v is reachable from vertex u in G'.*

Proof. Let G be a DAG with n vertices and m sources $s_1, s_2, \ldots, s_m$. We construct a new DAG G' as follows. We introduce a new source vertex s, and for each $i \in [1, m]$, add a directed path of length $i \cdot n$ from s to s_i. These paths are disjoint and use fresh dummy vertices not in G. The resulting graph G' is clearly acyclic and has a single source s. This transformation can be carried out in logspace.

Importantly, this construction:

- Preserves all paths and max-K-ness properties among the original vertices of G,
- Does not introduce any new paths or remove/alter any existing paths between internal vertices of G,
- Maintains reachability relations between G's vertices.
- Maintains longest paths and shortest paths between G's vertices.

We now prove that G is max-K if and only if G' is max-K. We consider both directions.

If G is not max-K, then there exist vertices $u, v \in V(G)$ such that there are at least $K+1$ distinct longest paths from u to v in G. Since our construction does not alter or add or remove any such paths, the same ambiguity persists in G', so G' is also not max-K.

If G is max-K, we consider all possible pairs $(u, v) \in V(G') \times V(G')$, and argue that in each case, the number of longest paths from u to v (if any path exists) is at most K.

- **Case 1:** $u, v \in V(G)$**.** Since G is max-K and the construction does not add/remove/alter any paths between original vertices, the number of longest paths from u to v in G' is at most K.
- **Case 2:** $u \in V(G), v \notin V(G)$**.** No such path exists in G', by construction.
- **Case 3:** $u \notin V(G), v \in V(G)$**.**
  - **Case 3.1:** $u \neq s$**.** Then u is a dummy vertex on the path from s to some source s_i. Any path from u to v must pass through s_i, and the prefix from u to s_i is fixed. If there exist $K+1$ longest paths from u to v, they must diverge after s_i, implying $K+1$ longest paths from s_i to v, contradicting the max-K-ness of G.
  - **Case 3.2:** $u = s$**.** Any path from s to v must pass through exactly one source s_i in G. Suppose there are two longest paths via distinct sources s_p and s_q, with $p < q$. Then:

    $$\text{Length from } s \text{ to } s_p = p \cdot n, \quad \text{Length from } s \text{ to } s_q = q \cdot n.$$

    Since any suffix path from s_i to v has length at most $n-1$, the longest such path from s to v through s_p lies in $[p \cdot n, p \cdot n + n - 1]$, and the longest such path from s to v through s_q lies in $[q \cdot n, q \cdot n + n - 1]$. Because $q \cdot n \geq (p+1) \cdot n > p \cdot n + n - 1$, the longest path through s_q strictly dominates, and hence two paths from different sources cannot have equal length.

    If all longest paths go through the same source s_i, they must diverge after s_i, which means there can be at most K longest paths from u to v since G is max-K.
- **Case 4:** $u, v \notin V(G)$**.** In this case, both vertices lie on some (possibly the same) prefix path. Since each path from s to s_i is unique and disjoint from others, at most one path can exist between any such pair (u, v).

Thus, in all cases, max-K-ness is preserved. ◀

Based on the above reduction, we get the following three lemmas.

▶ **Lemma 6.** *The problem of deciding max-K-ness of DAGs is logspace reducible to the problem of deciding max-K-ness of single-source DAGs.*

▶ **Lemma 7.** *The problem of deciding reachability in a max-K DAG is logspace reducible to the problem of deciding reachability in a single-source max-K DAG.*

▶ **Lemma 8.** *The problem of finding longest paths between two vertices in a max-K DAG is logspace reducible to the problem of finding longest paths between two vertices in a single-source max-K DAG.*

5 Deciding Max-K-ness of DAGs

In this section, we describe an unambiguous $O(\log K + \log n)$ space algorithm to decide whether a given DAG G is max-K. In fact, we can also make this algorithm work even without any promise on the input of being a DAG. If the input is not a DAG, our algorithm rejects along all computation paths. Hence, we get the following theorem.

▶ **Theorem 9.** *The set of max-K DAGs can be recognized in unambiguous $O(\log K + \log n)$ space.*

Proof. Let us consider that we are given a graph G and we need to decide if G is a max-K DAG or not. We can construct a single-source graph G' from G such that G' is a max-K DAG if and only if G is a max-K DAG (as shown in Theorem 5). First, we apply our algorithm (from Theorem 1) to decide if a given single-source graph is a single-source source-rooted max-K DAG on G'. If there is no accepting computation path, it will imply that either G is not acyclic or G is not max-K. In other words, G is not a max-K DAG. If there is some accepting computation path, we are sure that G' is a single-source DAG max-K with respect to the source. However, this does not automatically imply that G is max-K. Now, as shown in Section 3, we can reduce reachability between any two vertices in a single-source DAG G' to finding longest path from the source to some vertex in a single-source DAG G'', such that G'' is max-K if G' is max-K. We use this transformation. We check if G'' is a single-source DAG max-K with respect to the source (Theorem 1). If not, we know that G' is not max-K and in that case, our algorithm will also reject along all computation paths. Otherwise, there will be a unique computation path that will assert that G'' is indeed max-K with respect to its source. Once we are confirmed about that, we can find the longest path from the source to any other vertex in G'' (Theorem 1), which then also gives us the answer to the reachability question in G'. Now, we can check whether a subgraph having a particular vertex as the source is source-rooted max-K or not, using reachability in G'. In order to decide whether a subgraph is max-K or not, we check whether for every vertex $u \in V(G')$, the subgraph induced by all vertices reachable from u is source-rooted max-K (Theorem 1). If this holds for every u, then G' is globally max-K, otherwise,not. If G' is max-K then each call to reachability will give the correct answer along exactly one computation path. Therefore, the subgraph rooted at a particular vertex will be derived along exactly one computation path. Now, the routine to check source-rooted max-uniqueness will also accept each such subgraph along exactly one computation path. Thus, the computation will always proceed along one computation path and finally accept. On the other hand, if G' is not max-unique, then some oracle calls may reject along all computation paths and the computation will not proceed further. Thus, max-K-ness of G' can be decided in unambiguous $O(\log K + \log n)$ space as all the steps can be performed within this bound. ◀

▶ Remark 10. The algorithm described in this section doesn't prove that max-K-ness of a DAG can be decided in co-unambiguous $O(\log K + \log n)$ space. This is because the routine used here to check whether a graph is a single-source DAG max-K with respect to the source or not is not co-unambiguous (for reasons discussed in Section 2).

We can also show that deciding reachability and finding the length of the longest path between any two vertices in a max-K DAG can be done in unambiguous and co-unambiguous space $O(\log K + \log n)$

▶ **Theorem 11.** *Given a max-K DAG G and two vertices s and t in G, it can be decided whether t is reachable from s and the length of the longest path from s to t can be found in unambiguous and co-unambiguous $O(\log K + \log n)$ space.*

Proof. We construct graph G' from G as in Theorem 5. If G is a max-K DAG, G' is guaranteed to be a single-source max-K DAG. Moreover, for any two vertices u and v in G, v is reachable from u in G if and only if v is reachable from u in G' and the length of the longest path from u to v in G is the same as the length of the longest path from u to v in G'. Now, we can decide reachability between any two vertices in G', by Theorem 4. By computing reachability from s to all other vertices in G', we compute the subgraph induced by all vertices reachable from s and then in this single-source max-K subgraph, we can find the length of the longest path from s to t, if any path exists in unambiguous and co-unambiguous $O(\log K + \log n)$ space (Theorem 2). ◀

6 Para-UL Algorithm for Long Path

In this section, we will show that given a DAG G and two vertices s and t, it can be decided whether there is a path of length at least $n - k$ from s to t in unambiguous $O(k + \log n)$ space. Also, we don't require any promise on the input to be a DAG since the algorithm will reject any input that is not a DAG. Thus, the problem of deciding whether there is a path of length at least $n - k$ between two vertices in a DAG is in para-UL in terms of the parameter k.

In order to prove this result, we first state the following lemma.

▶ **Lemma 12.** *Let $G = (V, E)$ be a directed acyclic graph on n vertices. If G contains a directed path of length $n - k$, then for any pair of vertices $u, v \in V$, the number of longest paths from u to v in G is at most 2^k.*

Proof. Let P be a path of length $n - k$ in G. Define

$$S := \{v \mid v \in P\}, \qquad X := V(G) \setminus S.$$

Let $u, v \in V(G)$, and let P_1, P_2 be two longest paths from u to v that uses exactly the same set of vertices X' out of all the vertices in X.

P_1 and P_2 visit the vertices of X' in the same order. If not, then for some $x, y \in X'$, one path visits x before y and the other visits y before x. This yields directed paths $x \rightsquigarrow y$ and $y \rightsquigarrow x$, forming a cycle, a contradiction.

Let the common order be

$$x_1, x_2, \ldots, x_r,$$

and define $x_0 := u$, $x_{r+1} := v$.

Both P_1 and P_2 decompose into subpaths between consecutive vertices from X':

$$x_0 \rightsquigarrow x_1, \quad x_1 \rightsquigarrow x_2, \quad \ldots, \quad x_r \rightsquigarrow x_{r+1},$$

where all internal vertices lie in S.

Now we will show that for every i, the $x_i \rightsquigarrow x_{i+1}$ subpaths in P_1 and P_2 are identical. We will prove this by contradiction. Let S_1 and S_2 be the subpaths connecting $x_i \rightsquigarrow x_{i+1}$ in P_1 and P_2 respectively. Let us assume $S_1 \neq S_2$.

First, S_1 and S_2 must have the same length. Otherwise, suppose without loss of generality that S_1 has length l_1 and S_2 has length l_2 with $l_1 < l_2$. Let P_1 have total length l. Then the portion of P_1 outside S_1 has length $l - l_1$. Replacing S_1 in P_1 with S_2 yields a directed u-v path of length

$$(l - l_1) + l_2 > l,$$

contradicting the assumption that P_1 is a longest u-v path.

Thus, S_1 and S_2 have equal length, say l'. Let v_1 and v_2 be the first vertices where they differ. Since $v_1, v_2 \in S$, and S lies on P, either there exists a path from v_1 to v_2 or from v_2 to v_1. Assume without loss of generality that there is a path from v_1 to v_2.

Construct a walk by following S_1 from x_i to v_1, then the subpath of P from v_1 to v_2, and then S_2 from v_2 to x_{i+1}. Each concatenation is valid since the endpoint of one segment coincides with the start of the next, so this forms a directed walk; as G is acyclic, it is in fact a simple path. Since $v_1 \neq v_2$, the subpath from v_1 to v_2 has length $h \geq 1$, and hence the total length of this path is $j + h + (l' - j) = l' + h > l'$. Replacing S_1 in P_1 with this longer subpath produces a u-v path longer than P_1, a contradiction.

Therefore, each segment between two consecutive vertices from X in a longest path from u to v is uniquely determined. Hence, for each subset $X' \subseteq X$, there is at most one longest u-v path using exactly the vertices in X'.

Since $|X| = k$, there are at most 2^k such subsets. Therefore, the number of longest u-v paths is at most 2^k. ◀

Now, we will prove the following theorem.

▶ **Theorem 13.** *Given a DAG G and two vertices s and t in G, it can be decided in unambiguous $O(k + \log n)$ space, if there exists a path of length at least $n - k$ from s to t.*

Proof. Given a graph G, we first check if it is a max-2^k DAG or not in unambiguous $O(k + \log n)$ space Theorem 9. If G is not a max-2^k DAG, then by Lemma 12 there cannot be a path of length at least $n - k$ from s to t in G and our algorithm also rejects along all computation paths in this case. If G is a max-2^k DAG, then our algorithm asserts that along a unique computation path. Now, since we know G is a max-2^k DAG, we find the length of the longest path from s to t (if any path exists) in unambiguous and co-unambiguous $O(k + \log n)$ space Theorem 11. If the length is at least $n - k$, we accept. Otherwise, if the length is less than $n - k$ or t is not reachable from s, we reject. Thus, we can decide if there exists a path of length at least $n - k$ between any two given vertices in a DAG in unambiguous $O(k + \log n)$ space. ◀

7 X(UL ∩ coUL)P Algorithm for Long Path

In [2], it was shown that given a DAG G and two vertices s and t, it can be decided whether there is a path of length at least $n - k$ from s to t in unambiguous and co-unambiguous $O(k \log n)$ space and $O(n^{k+3})$ time (assuming word RAM model with $O(\log n)$ word size). Although polynomial in n, however this bound is not polynomial in k. In this section, we will show that the same problem can be solved in unambiguous and co-unambiguous $O(k \log n)$ space and $O(n^3 k^2)$ time (assuming word RAM model with $O(\log n)$ word size), thus making the time bound polynomial in both n and k. Also, we ensure that our algorithm works even without any promise on the input to be a DAG, such that our algorithm rejects any input that is not a DAG.

▶ **Theorem 14.** *Given a DAG G and two vertices s and t in G, it can be decided whether there exists a path of length at least $n - k$ from s to t in unambiguous and co-unambiguous $O(k \log n)$ space and simultaneous polynomial (in both n and k) time.*

Proof. Let us first discuss some structural properties of DAGs before going into the algorithm.

A *layering* of a directed acyclic graph (DAG) $G = (V, E)$ is a partition of V into numbered subsets $L_0, L_1, \ldots$ such that every edge of G goes from a vertex in a lower-numbered subset to a vertex in a higher-numbered subset. Let G be a DAG and let $v \in V$. We define a

layering such that vertex v belongs to *layer* i if the length of the longest path from any source of G to v is exactly i. With this definition, the sets of vertices belonging to layers $0, 1, 2, \ldots$ form a valid layering of the graph and we get the following lemma.

▶ **Lemma 15.** *Let $G = (V, E)$ be a DAG, and define layers as follows: a vertex v is in layer i if and only if the length of the longest path from any source of G to v is exactly i. Then a vertex v is in layer i if and only if:*

- *all in-neighbors of v lie in layers numbered less than i, and*
- *there exists an in-neighbor of v that lies in layer $i - 1$.*

Proof. Suppose v lies in layer i. Then the longest path from any source to v has length exactly i. Let $(u, v) \in E$ be any incoming edge. Then any path to u can be extended by (u, v) to form a path to v. Hence the longest path to u must have length at most $i - 1$, since otherwise we would obtain a path to v of length greater than i, a contradiction. Therefore every in-neighbor u lies in some layer $j \leq i - 1$, i.e., in a layer numbered less than i. Now consider a longest path of length i from some source to v. Let u be the penultimate vertex on this path. Then $(u, v) \in E$, and the longest path to u has length exactly $i - 1$. Hence u lies in layer $i - 1$. This shows that there exists an in-neighbor of v in layer $i - 1$.

Suppose that all in-neighbors of v lie in layers numbered less than i, and that there exists an in-neighbor u of v in layer $i - 1$. Since u lies in layer $i - 1$, there exists a path of length $i - 1$ from some source to u. Appending the edge (u, v) gives a path of length i to v. Thus the longest path to v has length at least i. On the other hand, since all in-neighbors of v lie in layers at most $i - 1$, any path to v must pass through some in-neighbor x whose longest path from a source has length at most $i - 1$. Hence any path to v has length at most i, because otherwise we would obtain a path to x longer than its longest path. Therefore the longest path to v has length at most i. Combining both bounds, the longest path to v has length exactly i, and hence v lies in layer i. ◀

We call a layer *unambiguous* if it contains exactly one vertex, and *ambiguous* if it contains more than one vertex. Now, we state the following lemmas regarding the structure of a DAG containing n vertices and a path of length at least $n - k$.

▶ **Lemma 16.** *If a DAG G on n vertices contains a path of length at least $n - k$, the total number of vertices belonging to ambiguous layers is at most $2k$.*

Proof. Observe that if the DAG contains a path of length at least $n - k$, then the graph must contain at least $n - k$ layers, since the vertices of any path must belong to distinct layers. Let P be such a path. The vertices of P occupy $n - k$ distinct layers, and the remaining k vertices of the graph may belong either to these layers or to other layers. A layer becomes ambiguous when one or more of these additional vertices share a layer with a vertex of P or with each other. Each ambiguous layer must contain at least one vertex not on P. In the worst case, all of the k vertices not belonging to P may lie in ambiguous layers. Moreover, a vertex of P can lie in an ambiguous layer only if another vertex shares that layer. Since there are only k vertices outside P, there can be at most k vertices of P that lie in ambiguous layers. Consequently, the total number of vertices belonging to ambiguous layers is at most $2k$. ◀

▶ **Lemma 17.** *Given a DAG G, if v belongs to an unambiguous layer $j < i$, then it must occur as the j-th vertex on every path of length i to vertices in layer i.*

Proof. The length of the longest path from any source to v is exactly j. Because the layer is unambiguous, v is the unique vertex in layer j.

Now fix a vertex v_i in layer i. By definition of layering, there exists a path P of length i from some source to v_i. Consider the sequence of vertices along P:

$$s = v_0 \to v_1 \to \cdots \to v_i$$

We claim that v_ℓ lies in layer ℓ for each $0 \le \ell \le i$. Indeed, since P has length i, the prefix $v_0 \to \cdots \to v_\ell$ is a path of length ℓ to v_ℓ, so the longest path to v_ℓ has length at least ℓ. On the other hand, if v_ℓ had a path of length greater than ℓ from some source, then by appending the suffix of P from v_ℓ to v, we would obtain a path to v of length greater than i, contradicting that v lies in layer i. Hence v_ℓ lies in layer ℓ. In particular, v_j lies in layer j. Since layer j is unambiguous, it contains exactly one vertex, namely v. Therefore $v_j = v$, and hence v appears as the j-th vertex on path P.

Since P was an arbitrary path of length i to an arbitrary vertex in layer i, it follows that every such path must contain v as its j-th vertex. ◀

▶ **Lemma 18.** *Given a DAG G on n vertices containing a path of length at least $n-k$. Let v be a vertex in layer i of G. In any longest path from a vertex s to v in G, the predecessor of v lies either in layer $i-1$ or in an ambiguous layer numbered less than i.*

Proof. Let u be the predecessor of v in a longest s–v path, and suppose u lies in layer $j < i-1$ and is in an unambiguous layer. By Lemma 17, the j-th vertex in any longest path of length i from any source to v must be u. Since v lies in layer i, there exists a path from u to v of length $i-j > 1$. Replacing the edge (u, v) with the path from u to v of length $i-j$ yields a longer s–v path, a contradiction. Hence u must lie either in layer $i-1$ or in an ambiguous layer. ◀

We give a nondeterministic algorithm that inductively computes the layers of the DAG. It is a modification of the *inductive tracing algorithm* of [2], which decides whether there is a Hamiltonian (n-length) path from s to t in a DAG. That algorithm computes the i-th layer in iteration i and proceeds only if the layer is unambiguous. Given the unique vertex v_i, it searches for an out-neighbor u whose in-neighbors all lie in layers numbered $< i$. This is done by nondeterministically guessing a length-i path to v_i, counting the number of in-neighbors of u on this path and checking if this count matches the indegree of u. Since each layer up to i is unambiguous, this path is unique and contains all vertices from earlier layers, ensuring correctness and unambiguity.

We extend this approach to detect paths of length $n-k$. Now layers up to i need not be unambiguous, so vertices from earlier layers may not appear on every length-i path to layer i. To handle this, we maintain a list L of vertices in ambiguous layers seen so far (at most $2k$ vertices) and a set U of vertices in the current layer i.

Unlike the Hamiltonian case, there may be multiple length-i paths to a vertex in layer i, so naive guessing breaks unambiguity. To restore it, we use an inductive-counting–type idea: define the *weight* of a path as the integer obtained by concatenating the labels of vertices from ambiguous layers along the path, in order. We then get the following lemma.

▶ **Lemma 19.** *Given a DAG G and a vertex v in layer i of G, there is a unique minimum-weight path of length i to v corresponding to the weight function defined above.*

Proof. We can interpret the weight as a number in base n, where $n = |V|$, so that the comparison of weights corresponds to lexicographic comparison of the sequences of ambiguous-layer vertices.

First of all, any path of length i to v must have exactly one vertex from each layer $j < i$. All such paths agree on the vertices that lie in unambiguous layers since by Lemma 17, if a layer $j < i$ is unambiguous, then the unique vertex in that layer must appear as the j-th vertex on every path of length i to v. Therefore, any two such paths can differ only in the vertices chosen at ambiguous layers.

Thus, each path can be uniquely represented by the sequence of vertices it selects at ambiguous layers. The weight function assigns to each path exactly this sequence, encoded as an integer in base n. Since different sequences correspond to different integers, distinct paths yield distinct weights.

It follows that the set of weights corresponding to all length-i paths to v is a finite set of distinct integers. Therefore, there exists a unique minimum element in this set. The corresponding path is the unique minimum-weight path to v. ◀

For each vertex in the current layer U, we maintain w_U as the sum of the weights of the minimum-weight length-i paths to the vertices in U. When we guess length-i paths to the vertices in U, we ensure that the total weight equals w_U. Since only one computation path guesses all the minimum weight paths correctly, therefore only one computation path proceeds after each round of guessing.

So far this determines whether a path of length at least $n - k$ reaches t. To ensure it originates at s, we additionally store, for each vertex in $L \cup U$, the length of the longest path from s to that vertex.

We provide below the pseudo-code of the algorithm.

The algorithm uses three subroutines: AllInNeighborsCertified, Guess, and Weight. The subroutine Guess guesses a path to a vertex v, computes its weight, and counts edges to a vertex u from vertices on the path or in L. Using Guess, Weight computes the weight of the minimum-weight path to v, while AllInNeighborsCertified checks whether all in-neighbors of v lie on minimum-weight longest paths to vertices in U. Using these, Algorithm 1 inductively computes layers and decides reachability of t from s in G. We use "**return** ?" in Guess and Weight to denote incorrect nondeterministic guesses; such computation paths halt immediately. For brevity, we omit explicit propagation of "?" in the pseudocode. The algorithm returns a non-? value along a unique computation path, and hence decides both yes and no instances unambiguously.

▶ **Lemma 20.** *After each iteration i of the while loop (line 19) in Algorithm 1, U is exactly the set of vertices in layer i, L contains all vertices in ambiguous layers numbered at most i, w_U equals the sum of minimum weights of length-i paths to vertices in U, and for each vertex in $L \cup U$ we store the length of the longest path from s.*

Proof. We argue by induction on i. Assume correctness at the start of iteration i.

By Lemma 15, a vertex v belongs to layer $i + 1$ iff it has an in-neighbor in U and all its in-neighbors are in layers numbered $< i + 1$. A vertex u in an ambiguous layer $< i + 1$ lies in L. If u is in an unambiguous layer $< i + 1$, then by Lemma 17 it lies on every length-i path to layer i. Thus, in order to decide if v belongs to layer $i + 1$, we just need to check if it has an in-neighbor in U and if all its in-neighbors are either in L or belong to an i-length path to some vertex in U. This is exactly the check performed by AllInNeighborsCertified. The algorithm therefore computes U' correctly and updates L when the layer is ambiguous.

For w_U, fix $v \in U'$. The subroutine Weight considers all length-i paths to predecessors of v in U. By induction, w_U equals the sum of the weights of minimum-weight i-length paths to vertices in U. The check $w = w_U$ ensures only guesses corresponding to minimum-weight paths survive, and the minimum among extensions is selected. Thus Weight returns the

Algorithm 1 Decide whether there is an s–t path of length at least $n-k$ in a DAG.

Require: DAG $G=(V,E)$, vertices s,t, parameter k

Find the sources of G: $s_1, s_2, \ldots, s_m$
if $m > k+1$ **then**
    return reject
Initialize $L \leftarrow \emptyset$, $U \leftarrow \emptyset$, $w_U \leftarrow 0$, $r_t \leftarrow -1$, $num \leftarrow m$
if $m > 1$ **then**
    for $j = 1$ to m **do**
        if $s_j = s$ **then**
            Add $(s_j, 0)$ to L
        else
            Add $(s_j, -1)$ to L
for $j = 1$ to m **do**
    if $s_j = s$ **then**
        Add $(s_j, 0)$ to U
    else
        Add $(s_j, -1)$ to U
    if $m > 1$ **then**
        $w_U \leftarrow w_U + s_j$
Initialize $i \leftarrow 1$
while $U \neq \emptyset$ **do**
    $U' \leftarrow \emptyset$
    $w_{U'} \leftarrow 0$
    for each $v \in V$ such that $(u,v) \in E$ for some $(u,*) \in U$ **do**
        $r \leftarrow -1$
        if AllInNeighborsCertified(v, L, U, w_U, i) **then**
            $w_{U'} \leftarrow w_{U'} +$ Weight(v, L, U, w_U, i)
            for each $(z, r') \in L \cup U$ such that $(z,v) \in E$ **do**
                if $r' \geq 0$ and $r'+1 > r$ **then**
                    $r \leftarrow r'+1$
            if $v = s$ **then**
                $r \leftarrow 0$
            Add (v, r) to U'
            if $v == t$ **then**
                $r_t \leftarrow r$
            $num \leftarrow num + 1$
    $U \leftarrow U'$
    $w_U \leftarrow w_{U'}$
    $i \leftarrow i+1$
    if $|U| > 1$ **then**
        $L \leftarrow L \cup U$
    else
        $w_U \leftarrow w_U - (w_U \mod n)$
if $num \neq n$ **then**
    return reject
if $r_t \geq n-k$ **then**
    return accept
else
    return reject

Algorithm 2 Check whether all in-neighbors of v lie on minimum-weight longest paths.

```
1:  function AllInNeighborsCertified(v, L, U, w_U, j)
2:      Initialize indeg_v ← 0, flag ← 0, weight ← 0
3:      for each x such that (x, v) ∈ E do
4:          indeg_v ← indeg_v + 1
5:      for each (y, ∗) ∈ U do
6:          (w, count) ← Guess(v, y, L, j)
7:          weight ← weight + w
8:          if count = indeg_v then
9:              flag ← 1
10:     if weight ≠ w_U then
11:         return ?
12:     if flag = 0 then
13:         return false
14:     return true
```

Algorithm 3 Compute the minimum weight path to v.

```
1:  function Weight(v, L, U, w_U, j)
2:      Initialize w ← 0, weight ← ∞
3:      for each (x, r) ∈ U do
4:          (w′, ∗) ← Guess(x, x, L, j)
5:          w ← w + w′
6:          if (x, v) ∈ E and weight > w′ · n + v then
7:              weight ← w′ · n + v
8:      if w ≠ w_U then
9:          return ?
10:     return weight
```

Algorithm 4 Guess a path and compute its weight.

```
1:  function Guess(v, u, L, j)
2:      Initialize h ← 0, weight ← 0, count ← 0
3:      while h < j do
4:          Guess a vertex y ∈ V
5:          if h ≠ 0 and (x, y) ∉ E then
6:              return ?
7:          if (y, ∗) ∈ L then
8:              weight ← weight · n + y
9:          else if (y, v) ∈ E then
10:             count ← count + 1
11:         x ← y, h ← h + 1
12:     if x ≠ u then
13:         return ?
14:     for (y, ∗) ∈ L such that (y, v) ∈ E do
15:         count = count + 1
16:     return (weight, count)
```

minimum weight of a length-$(i+1)$ path to v. Summing over $v \in U'$ yields $w_{U'}$. If $|U'| = 1$, the adjustment $w_{U'} - (w_{U'} \bmod n)$ discards the last contribution, maintaining consistency with the encoding.

By Lemma 18, the penultimate vertex of a longest s–v path lies either in L or in U. Hence the length of the longest path to v is obtained by maximizing over the length of the longest paths to the in-neighbors of v lying in L or U. ◀

▶ **Lemma 21.** *Algorithm 1 returns a non-? value along exactly one computation path.*

Proof. For each $v \in U$ at iteration i, there is a unique minimum-weight length-i path (Lemma 19). The subroutines AllInNeighborsCertified and Weight accept iff all calls to Guess guess these paths correctly, enforced by the check $w = w_U$. Hence exactly one computation path survives in each iteration, and the algorithm returns a non-? value along a unique path deciding whether a path of length $\geq n - k$ exists or not. ◀

Finally, we show rejection on non-DAG inputs. We prove by induction that no vertex on a directed cycle ever enters U. Initially, U contains only sources. Suppose the following claim holds up to iteration i: every vertex in U is reachable from some source by a path of length i, and every vertex on any such path appeared in U in some earlier iteration and is not part of any directed cycle. Consider a vertex v that is added to U during iteration $i+1$. By the certification procedure, all in-neighbors of v must either belong to L (and hence appeared in U during some earlier iteration) or lie on one of the paths leading to vertices currently in U. Therefore, by our induction hypothesis, every in-neighbor of v lies on a path from a source that contains only vertices that appeared in earlier iterations. Consequently, any path to v must extend one of these paths, and therefore cannot contain a vertex that lies on a directed cycle. This shows that no vertex that belongs to a cycle can ever appear in U and any path to a vertex in U cannot contain a vertex that belongs to a cycle. Thus no cycle vertex ever appears in U, and vertices on cycles are never counted. Since num counts vertices added to U, $num < n$ implies the presence of a cycle, and the algorithm rejects.

The space usage is $O(|L|) = O(k \log n)$. If $|L| > 2k$, the algorithm rejects by Lemma 16. In the word RAM model, Guess runs in $O(nk)$ time. Both AllInNeighborsCertified and Weight invoke it $O(k)$ times, giving $O(nk^2)$ time. Each iteration processes all $v \in V$, yielding $O(n^2k^2)$ time per iteration and $O(n^3k^2)$ overall. ◀

References

1 Eric Allender. Guest column: Parting thoughts and parting shots (read on for details on how to win valuable prizes! *SIGACT News*, 54(1):63–81, 2023. doi:10.1145/3586165.3586175.

2 Ronak Bhadra and Raghunath Tewari. Inductive tracing and the complexity of finding hamiltonian path in dags. In Artur Jeż and Jan Otop, editors, *Fundamentals of Computation Theory - 25th International Symposium, FCT 2025, Wrocław, Poland, September 15-17, 2025, Proceedings*, volume 16106 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2025. doi:10.1007/978-3-032-04700-7_5.

3 Hans L. Bodlaender, Carla Groenland, Jesper Nederlof, and Céline M. F. Swennenhuis. Parameterized problems complete for nondeterministic FPT time and logarithmic space. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 193–204. IEEE, 2021. doi:10.1109/FOCS52979.2021.00027.

4 Anant Dhayal, Jayalal Sarma, and Saurabh Sawlani. Min/max-poly weighting schemes and the NL versus UL problem. *ACM Trans. Comput. Theory*, 9(2):10:1–10:25, 2017. doi:10.1145/3070902.

5 Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the space and circuit complexity of parameterized problems: Classes and completeness. *Algorithmica*, 71(3):661–701, 2015. doi:10.1007/S00453-014-9944-Y.

6 Jörg Flum and Martin Grohe. Describing parameterized complexity classes. *Inf. Comput.*, 187(2):291–319, 2003. doi:10.1016/S0890-5401(03)00161-5.

7 Neil Immerman. Nondeterministic space is closed under complement. *SIAM Journal on Computing*, 17:935–938, 1988. doi:10.1137/0217058.

8 Nutan Limaye, Meena Mahajan, and Prajakta Nimbhorkar. Longest paths in planar dags in unambiguous logspace. In Rod Downey and Prabhu Manyem, editors, *Theory of Computing 2009, Fifteenth Computing: The Australasian Theory Symposium, CATS 2009, Wellington, New Zealand, January 2009*, volume 94 of *CRPIT*, pages 99–105. Australian Computer Society, 2009. URL: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV94Limaye.html.

9 Aduri Pavan, Raghunath Tewari, and N. V. Vinodchandran. On the power of unambiguity in log-space. *Computational Complexity*, 21(4):643–670, 2012. doi:10.1007/s00037-012-0047-3.

10 K. Reinhardt and E. Allender. Making nondeterminism unambiguous. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 244–253, 1997. doi:10.1109/SFCS.1997.646113.

11 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

12 Robert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988. doi:10.1007/BF00299636.

13 Seinosuke Toda. On the complexity of topological sorting. *Inf. Process. Lett.*, 35(5):229–233, 1990. doi:10.1016/0020-0190(90)90050-8.