Min/Max-Poly Weighting Schemes and the NL vs UL Problem*

Anant Dhayal† Jayalal Sarma† Saurabh Sawlani†

May 3, 2016

Abstract

For a graph $G(V, E)$ ($|V| = n$) and a vertex $s \in V$, a weighting scheme ($w : E \to \mathbb{N}$) is called a *min-unique* (resp. *max-unique*) weighting scheme, if for any vertex v of the graph G, there is a unique path of minimum (resp. maximum) weight[1] from s to v. Instead, if the number of paths of minimum (resp. maximum) weight is bounded by n^c for some constant c, then the weighting scheme is called a *min-poly* (resp. *max-poly*) weighting scheme.

In this paper, we propose an unambiguous non-deterministic log-space (UL) algorithm for the problem of testing reachability graphs augmented with a *min-poly* weighting scheme. This improves the result due to Allender and Reinhardt[12] where a UL algorithm was given for the case when the weighting scheme is *min-unique*.

Our main technique involves triple inductive counting, and generalizes the techniques of [8, 13] and [12], combined with a hashing technique due to [6] (also used in [7]). We combine this with a complementary unambiguous verification method, to give the desired UL algorithm.

At the other end of the spectrum, we propose a UL algorithm for testing reachability in layered DAGs augmented with *max-poly* weighting schemes. To achieve this, we first reduce reachability in layered DAGs to the longest path problem for DAGs with a unique source, such that the reduction also preserves the *max-unique* and *max-poly* properties of the graph. Using our techniques, we generalize the double inductive counting method in [9] where UL algorithm was given for the longest path problem on DAGs with a unique sink and augmented with a *max-unique* weighting scheme.

An important consequence of our results is that, to show $\mathsf{NL} = \mathsf{UL}$, it suffices to design log-space computable *min-poly* (or *max-poly*) weighing schemes for layered DAGs.

1 Introduction

Reachability testing in graphs (Reach) is an important algorithmic problem that encapsulates central questions in space complexity. Given a graph $G(V, E)$ and two special vertices s and t, the problem asks to test if there is a path from s to t in the graph G. The problem admits a (deterministic) log-space algorithm for the case of trees and undirected graphs (by a breakthrough result due to Reingold[11]). The directed graph version of the problem captures the complexity class NL. Designing a log-space algorithm for the problem is equivalent to proving $\mathsf{NL} = \mathsf{L}$. (See [1] for a survey.) Even in the the case when the graph is a layered DAG[2] ,the problem is known to be be NL-complete.

*The preliminary version of this work was presented at the conference FSTTCS 2014.

†Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India.

[1] Weight of a path p is the sum of the weights of the edges appearing in p.

[2] A DAG is layered, if V can be partitioned as $V = V_1 \cup \ldots V_\ell$ s.t. edges go from V_i to V_{i+1} for some i.

An important intermediate class of algorithms for reachability is when the non-determinism is unambiguous - when the algorithm 'accepts' in at most one of the non-deterministic paths. The class of problems which can be solved by such restricted non-deterministic algorithms using only log-space is called Unambiguous Log-space (UL). Under a non-uniform polynomial-sized advice, the reachability problem is known to have a UL algorithm[12], thus showing NL/poly = UL/poly. Central to arriving at this complexity theoretic result was the following algorithmic result that Allender and Reinhardt[12] had established: testing reachability in a graph G augmented with a log-space computable weighting scheme that maps $w : E \to \mathbb{N}$ such that there is a unique minimum-weight path from s to any vertex v in the graph, can be done by a non-deterministic log-space algorithm unambiguously and hence is in the complexity class UL. (We call such weighting schemes as *min-unique* weighting schemes.) This also led to other important developments including an unambiguous log-space algorithm for directed planar reachability [5] - which was achieved by designing a log-space computable *min-unique* weighting scheme for reachability in grid-graphs (a special class of planar graphs for which reachability is as hard as planar reachability[2]). An important open problem in this direction is to design a log-space *min-unique* weighting scheme for general graphs. The UL-computable version of this is also known to be equivalent to showing NL = UL.

Our Results: We make further progress on this algorithmic front by relaxing the restriction on the number of paths of minimum weight from one to polynomially many paths. We call a weighting scheme *min-poly* if it results in at most polynomially many (in terms of $n = |V|$) paths of minimum weight from s to any vertex v in a graph $G(V, E)$.

Theorem 1. *Testing reachability in graphs augmented with log-space computable* min-poly *weighting schemes, can be done by a non-deterministic log-space algorithm unambiguously and hence is in the complexity class* UL.

Our algorithms use the technique of triple inductive counting. The method of inductive counting was originally discovered and employed as an algorithmic technique in [8] and [13] in order to design non-deterministic log-space algorithms for testing non-reachability in graphs. A double inductive version of this was used again by Allender and Reinhardt[12] for designing unambiguous non-deterministic algorithms for testing reachability graphs augmented with *min-unique* weighting schemes. We use a triple inductive version of the inductive counting method, keeping track of one extra parameter (which is the sum of the number of minimum weight paths to each vertex). Along with a hashing technique (also used in [7]), this leads to a non-deterministic algorithm where each accepting configuration has at most one path leading to it on any input (the corresponding complexity class is known as FewUL). Finally, we convert this algorithm to a UL algorithm using an unambiguous complimentary verification technique, thus completing the proof of the theorem.

A natural complementary question is if similar complexity bounds hold in the case of graphs with weighting assignments that results in unique maximum weight paths from s to any vertex v (such weighting schemes are called *max-unique* weighting schemes). In [9], the longest path problem on DAGs augmented with *max-unique* weighting assignments and having a unique sink t, was shown to be in UL. The corresponding weighting scheme with polynomially many paths of maximum weight will be called a *max-poly* weighting scheme. Using our triple inductive and complementary verification techniques, we adapt their algorithms to improve their results by relaxing the constraint

on the weighting assignments - from *max-unique* to *max-poly*. We present our theorem in terms of the reachability problem, since we also show a reduction (Lemma 4.1) from the reachability problem on layered DAGs to the longest path problem on single source DAGs, where the *max-unique* and *max-poly* property of the graph is preserved.

Theorem 2. *Testing reachability in layered DAGs augmented with log-space computable* max-poly *weighting schemes, can be done by a non-deterministic log-space algorithm unambiguously and hence is in the complexity class* UL.

Observing that Theorem 1 and Theorem 2 hold even when the weighting schemes are UL-computable, and combining with the results of [10], it follows that: for any graph G there is a UL-computable *min-poly* weighting scheme if and only if there is a UL-computable *min-unique* weighting scheme. By a minor variant the proof technique in [10], it can be concluded (Proposition 5.1 in Section 5) that showing NL = UL is equivalent to designing UL-computable *(min)max-unique* weighting schemes which, thus, is equivalent to designing UL-computable *(min)max-poly* weighting schemes.

An important comparison of our results is with a complexity theoretic collapse result shown by [7]. FewL is the class of problems that has non-deterministic algorithms with only polynomially (in n) many accepting paths on any input of length n. Clearly, FewL contains all problems in UL - however, the converse is not known. In its algorithmic flavour, this question asks if reachability in a graph with at most polynomially many paths from s to t, can be done by a non-deterministic algorithm in log-space, producing at most one accepting path. ReachUL and ReachFewL are the corresponding complexity classes where the uniqueness and polynomially boundedness constraints are respectively applied for the number of paths from s to any other $v \in V$. Clearly, ReachUL is contained in ReachFewL and were recently shown to be equal [7]. It is worthwhile noting that this establishes an unambiguous log-space algorithm for reachability in graphs where there are only polynomially many paths from the start vertex to any vertex in the graph. The graphs that we discussed above (i.e. graphs augmented with *min/max-poly* weighting schemes) also include such graphs trivially by keeping 1 as the weight assignment (by the weighting schemes) to all the edges.

2 Preliminaries

In this section, we provide the definitions and results referred to in the paper. Additionally, the reader can refer [4] and [3] for any undefined or unexplained definitions, reductions or results.

A layered DAG is a directed acyclic graph (DAG) in which the vertices can be partitioned into ℓ ordered layers such that each edge moves from a vertex in layer j to one in layer $j+1$, for all $0 \leq j \leq \ell - 1$. In such DAGs, we can label the vertices with unique positive integers in such a way that the label for any vertex in layer j is less than that for a vertex in layer $j+1$, for all $0 \leq j \leq \ell - 1$. This way, if there is an edge from a vertex labelled u to a vertex labelled v, then $u < v$.

In the context of this paper, we will use the Turing machine as our computational model. We will define some of the relevant complexity classes with respect to the space and time used by a Turing machine to solve a certain problem.

Log-space (L) and *Non-deterministic log-space* (NL) are complexity classes of problems which can be solved by deterministic and non-deterministic Turing machines, respectively - where the length of the work tape is logarithmic in terms of the input length.

A problem is said to be in the class *Unambiguous log-space* (UL), if it can be solved by a non-deterministic log-space Turing machine such that, on any given input, at most one computational path leads to an 'accept' state. A problem is said to be in the class FewL, if it can be solved by a non-deterministic log-space Turing machine such that, on any input, at most polynomially many computational paths lead to an 'accept' configuration.

Another related complexity class is called FewUL. A problem is said to be in the class FewUL, if it can be solved by a non-deterministic log-space Turing machine such that, on any input *and a given 'accept' configuration* $\mathcal{C}$, at most one computational path leads to $\mathcal{C}$.

A language L is said to be in the complement class of some complexity class C (called co-C) if the language $\overline{L} \in C$. It is known [8, 13] that NL = co-NL.
The following containment can be drawn from the definitions itself:

$$\mathsf{L} \subseteq \mathsf{UL} \subseteq \mathsf{FewUL} \subseteq \mathsf{FewL} \subseteq \mathsf{NL}$$

We also formally define two problems which will appear in the paper:

$\textsc{Reach} = \{(G, s, t) \mid G \text{ contains a path from } s \text{ to } t\}$
$\textsc{LongPath} = \{(G, s, t, j) \mid G \text{ contains a simple path of length} \geq j \text{ from } s \text{ to } t\}$

Weighting Schemes: Given a graph $G(V, E)$, a *weighting scheme* $W : E \mapsto \mathbb{Z}^+$ is an assignment of positive integers, called *weights* to the edges of a given graph $G(V, E)$. The weight of an edge $e \in E$ is represented as $W(e)$. The weight of a path is defined as the sum of the weights of edges in the path. A weighting scheme on a graph $G(V, E)$ is called a *min-unique* (*max-unique*) weighting scheme if for every vertex $v \in V$, the number of minimum (maximum)-weight paths from s to v is at most one.

Similarly, a weighting scheme on a graph $G(V, E)$ is called a *min-poly* (*max-poly*) weighting scheme if for every vertex $v \in V$, the number of minimum (maximum)-weight paths from s to v is bounded by $|V|^c$, where c is a given constant.

When a particular weighting scheme is given or is computed for a graph G, rather than storing the weights as an appendix to the graph, we '*apply*' the weighting scheme to the graph in the following way: an edge (u, v) of weight w is replaced by a $u \rightsquigarrow v$ path of length w.

If a *min-unique* (*max-unique*) weighting scheme is 'applied' to a graph, then the resultant graph itself is known as a *min-unique* (*max-unique*) graph. Similarly, if a *min-poly* (*max-poly*) weighting scheme is 'applied' to a graph, then the resultant graph is known as a *min-poly* (*max-poly*) graph.

3 Reachability in *min-const* and *min-poly* graphs

In this chapter, we give an unambiguous non-deterministic log-space (UL) algorithm for Reach on *min-poly* graphs. In the process, we will first show a UL algorithm for Reach on *min-const* graphs[3] in an attempt to highlight the primary idea behind the algorithms.

3.1 A Warmup Case : UL algorithm for Reach on *min-const* graphs

In this section, we introduce a modification of Allender and Reinhardt's [12] algorithm for Reach on *min-unique* graphs, which allows it to work for *min-const* graphs. We will borrow the following terminology from their paper:

[3]Graphs in which the number of minimum-weight $s \rightsquigarrow t$ paths is bounded by a constant

Let v be a vertex of G. $d(v)$ is the length of the shortest $s \rightsquigarrow v$ path in G. Additionally, we define $p(v)$ as the number of shortest $s \rightsquigarrow v$ paths in G. Note that, in the case of *min-unique* graphs, $p(v)$ was at most 1. C_k is a set of vertices which is defined as follows: $C_k = \{v \mid d(v) < k\}$. The following variables are the ones computed inductively in the algorithm: $c_k = |C_k|$ and $\Sigma_k = \sum_{v \in C_k} d(v)$.

To handle a constant number of minimum-length paths, we introduce a new inductive parameter p_k which stores the sum of the number of minimum length paths from s to every vertex $v \in C_k$. Thus, $p_k = \sum_{v \in C_k} p(v)$.

Idea: First, we design an unambiguous log-space routine CHECK-MIN-CONST which returns $d(v) \leq k$ (in at most one non-deterministic path), assuming that the correct values of c_k, Σ_k and p_k are given.

Using CHECK-MIN-CONST as a subroutine, the routine UPDATE-MIN-CONST computes inductively the values of c_{k+1}, Σ_{k+1} and p_{k+1}, given that the input graph is *min-const* (considering vertices only in C_k), and the correct values of c_k, Σ_k and p_k are known. Here, we also check if there are more than the set constant number of paths to any vertex in $C_{k+1} - C_k$. If yes, then we reject.

We know that the input graph is *min-const* considering vertices in C_0 and we also know the correct values of c_0, Σ_0 and p_0. So, in MAIN-MIN-CONST we will use the above routines to compute c_{n-1}, Σ_{n-1} and p_{n-1} and verify that the graph is *min-const*. Once, this is done we will check whether t is reachable from s or not. If reachable, we will accept.

In this algorithm, the parameter p_k is also required because, now for each vertex v we have $p(v)$ minimum-length paths, and unlike in the case of *min-unique* graphs, now $p(v)$ can be more than one. So, while guessing the minimum-length path to any vertex v, we can have more than one non-deterministic path in the algorithm which comes up with valid minimum-length $s \rightsquigarrow v$ paths in G. So, now in our algorithm CHECK-MIN-CONST, every time we need to guess a minimum-length path to some vertex v, we will guess $p(v)$ and then guess $p(v)$ many minimum-length $s \rightsquigarrow v$ paths. Also, we will return the value $p(v)$ if $v \in C_k$, and 0 if $v \notin C_k$. Additionally, we will maintain an order while guessing the $p(v)$ paths, so that the guessing procedure remains unambiguous. Finally, we will verify that for each vertex v, the guessed value of $p(v)$ is indeed correct.

Guessing constant paths unambiguously in log-space: An ℓ length path π is said to be 'lexicographically greater' than another ℓ-length path π' if at the first vertex where both the paths differ (starting from the start vertex s), π has a vertex with higher label than π'.

The following demonstrates how the algorithm guesses q $s \rightsquigarrow v$ paths of length ℓ unambiguously in log-space.

- Set aside $q \log n + \log \ell + q - 1$ bits (all initialized to 0) from the empty part of the tape.
- The first q slices of $\log n$ bits each are meant to contain the current nodes of the q paths (all initialized to $\log n$ zeros, indicating node s).
- The next $\log \ell$ bits contain the counter of the current path length for all the q paths being guessed.
- Of the last $q - 1$ bits, the i^{th} bit indicates whether or not the i^{th} path is lexicographically greater than the $i + 1^{th}$ path (according to the guessing done so far).

- Non-deterministically guess the next node for each of the q paths. Each time, replace the current q nodes by the next q nodes, in the q slices. After every such guess increase the counter contained in the $\log \ell$ bits by 1. Reject if the nodes guessed do not signify valid paths.
- Update the last $q-1$ bits every time you guess a next node for some path. Make the i^{th} bit 1 at the very first time when the next node in the i^{th} path is lexicographically greater than that of the $i+1^{th}$ path. Once the i^{th} bit is made 1, the next nodes guessed in i^{th} and $i+1^{th}$ paths need not follow any order. But, until the i^{th} bit remains 0, same next nodes have to be guessed for both the paths.
- If, at any time and for any i, the $i+1^{th}$ path is lexicographically greater than the i^{th} path, reject.
- Before all the $\ell+1$ nodes have been guessed for all the paths, all the $q-1$ bits should become 1 (as we are guessing q distinct paths of length ℓ). Also, the $\ell+1^{th}$ node for all the paths should be v. In any other case, reject.

Algorithm The following is a UL algorithm to decide the language:

$$\textsc{Min-Const Reach}_c = \{(G(V,E),s,t) \mid \exists\ s \rightsquigarrow t \text{ path and } \forall v \in V, p(v) \leq c\}$$

Algorithm 3.1 MAIN-MIN-CONST: Main UL routine for MIN-CONST REACH$_c$

Input: (G,s,t)
$k := 1$
$c_0 := 1; \Sigma_0 := 0; p_0 := 1$
$(c_1, \Sigma_1, p_1) =$ UPDATE-MIN-CONST$(G, s, 0, c_0, \Sigma_0, p_0)$
while $k < n-1$ and $(c_{k-1}, \Sigma_{k-1}, p_{k-1}) \neq (c_k, \Sigma_k, p_k)$ **do**
  $(c_{k+1}, \Sigma_{k+1}, p_{k+1}) =$ UPDATE-MIN-CONST$(G, s, k, c_k, \Sigma_k, p_k)$
  $k := k+1$
end while
if CHECK-MIN-CONST$(G, s, t, k, c_k, \Sigma_k, p_k) > 0$ **then**
  ACCEPT
else
  REJECT
end if

Algorithm 3.2 UPDATE-MIN-CONST: Deterministic (barring CHECK-MIN-CONST calls) routine for computing c_{k+1}, Σ_{k+1} and p_{k+1}

Input: $(G, s, k, c_k, \Sigma_k, p_k)$
Output: $c_{k+1}, \Sigma_{k+1}, p_{k+1}$
$c_{k+1} := c_k$; $\Sigma_{k+1} := \Sigma_k$; $p_{k+1} := p_k$;
$num := 0$;
for $v \in V$ **do**
  if CHECK-MIN-CONST$(G, s, v, k, c_k, \Sigma_k, p_k) = 0$ **then**
    for x such that $(x, v) \in E$ **do**
      $num := num +$ CHECK-MIN-CONST$(G, s, x, k, c_k, \Sigma_k, p_k)$;
      if $num > c$ **then**
        REJECT
      end if
    end for
    if $num > 0$ **then**
      $c_{k+1} := c_{k+1} + 1; \Sigma_{k+1} := \Sigma_{k+1} + k + 1; p_{k+1} := p_{k+1} + num$;
    end if
  end if
end for

Algorithm 3.3 CHECK-MIN-CONST: UL routine to compute the number of paths to v

Input: $(G, s, v, k, c_k, \Sigma_k, p_k)$
$count := 0; sum := 0; paths := 0; paths.to.v := 0$;
for $x \in V$ **do**
  Non-deterministically guess $0 \leq q \leq c$
  if $q \neq 0$ **then**
    Non-deterministically guess $0 \leq \ell \leq k$
    Non-deterministically guess q paths $p_1, p_2, \ldots p_q$ of length exactly ℓ each from s to x.
    if (the paths are not valid) **then**
      REJECT
    end if
    $count := count + 1; sum := sum + \ell; paths := paths + q$;
    if $x = v$ **then**
      $paths.to.v := q$;
    end if
  end if
end for
if $count = c_k$, $sum = \Sigma_k$ and $paths = p_k$ **then**
  Return the value of $paths.to.v$
else
  REJECT
end if

Proof of Correctness:

Claim 3.1.1. *If G is* min-const *considering only vertices in C_k, and correct values of c_k, Σ_k and p_k are given, then the algorithm* Check-min-const *has exactly one non-rejecting path, and it returns the correct value of $p(v)$ if $d(v) \leq k$ and returns 0 if $d(v) > k$.*

Proof. We argue that $\forall x \in V$, there is a unique way to guess the values of q and ℓ, such that the equalities $count = c_k, sum = \Sigma_k$ and $paths = p_k$ are satisfied. We analyze this by cases :

- If the algorithm, in a non-deterministic path, guesses $q \neq 0$ (i.e. $d(x) \leq k$) for some vertex x for which $q = 0$ (i.e. $d(x) > k$), then it will not be able to guess any path of length $\leq k$, and hence will end up rejecting in that non-deterministic path. If it guesses $q = 0$ (i.e. $d(x) > k$) for some vertex x for which $q \neq 0$ (i.e. $d(x) \leq k$), it will have to guess $q \neq 0$ (i.e. $d(y) \leq k$) for some vertex y for which $q = 0$ (i.e. $d(y) > k$), and hence will reject in that non-deterministic path.
- Now, for the non-deterministic paths where $q \neq 0$, i.e. $x \in C_k$:
  - If the algorithm, in a non-deterministic path, guesses $\ell < d(x)$ for any vertex x, then it will not be able to find a path of such length and hence will end up rejecting in that non-deterministic path. If it guesses $\ell > d(x)$, then to compensate, it will have to guess $\ell < d(y)$ for some other vertex y, and hence will reject in that non-deterministic path.
  - If the algorithm, in a non-deterministic path, guesses $q > p(x)$ for any vertex x, then it will not be able to guess q many paths, and hence will end up rejecting in that non-deterministic path. If it guesses $q < p(x)$, then to compensate, it will have to guess $q > p(y)$ for some other vertex y, and hence will reject in that non-deterministic path.

Hence, only the path in which, for each vertex x, "$q = 0$?" (i.e. "$d(x) \leq k$?") is guessed correctly, and the guessed value of q is equal to $p(x)$ and ℓ is equal to $d(x)$ (if $q \neq 0$), will be a non-rejecting path and will return the correct value of $p(v)$ if $d(v) \leq k$ and 0 if $d(v) > k$ (as in this case the if-condition at line 5 will not be satisfied). □

Claim 3.1.2. *If G is* min-const *considering only vertices in C_k and correct values of c_k, Σ_k and p_k are given, the algorithm* Update-min-const *runs deterministically barring the calls to the routine* Check-min-const*, and computes the correct values of c_{k+1}, Σ_{k+1} and p_{k+1}. It also rejects if G is not* min-const *considering only vertices in C_{k+1}.*

Proof. The algorithm first assigns $c_{k+1} := c_k, \Sigma_{k+1} := \Sigma_k$ and $p_{k+1} = p_k$. Now, to update these values we need the exact set of vertices with $d(v) = k + 1$, i.e. the vertices in $C_{k+1} - C_k$. The algorithm, for each v, checks if $d(v) > k$ and for each of its neighbours x, calls Check-min-const. If all the calls return 0, i.e. no neighbour is in C_k, we do not do anything and move on to the next vertex. If the sum of the values returned by all the calls (which is being stored in the variable *num* in line 8 is in the range 1 to c, p_{k+1} is incremented by this value, c_{k+1} is incremented by 1 and Σ_{k+1} is incremented by $k + 1$ (line 14). If at any point on line 8, the variable *num* exceeds c we reject (line 10), as the input graph G is not min-const. Thus, all the three parameters get updated correctly and hence the proof. □

Observation 3.1. *Observe that, since G is* min-const *considering vertices in C_0 and we begin with the correct values of c_0, Σ_0 and p_0 (we assume that there is 1 zero length path from s to s), by induction, Claims 3.1.1 and 3.1.2 imply that the values of c_k, Σ_k and p_k calculated after every*

iteration of while-loop (lines 5 to 8) in the Main-Const *algorithm are correct. Also, the iteration ends without rejecting, if and only if the graph G is* min-const *considering only vertices in C_k.*

Lemma 3.1. *The algorithm* Main-Const *has at most one path to ACCEPT state. Additionally, the algorithm has exactly one path to ACCEPT state if and only if G is a* min-const *graph and t is reachable from s.*

Proof. Using Observation 3.1 and Claim 3.1.1, we know that there is exactly one non-rejecting path in each call to Check-min-const. Thus, there is exactly one non-rejecting path in each call to Update-min-const, as Update-min-const is deterministic barring the calls to Check-min-const. Similarly, there is exactly one non-rejecting path in Main-min-const, as Main-min-const is deterministic barring the calls to Update-min-const. If G is not *min-const*, one of the calls to Update-min-const (line 6) will lead in rejection. If G is *min-const*, the while-loop (lines 5 to 8) will terminate without rejecting. If t is indeed reachable from s, this non-rejecting path goes to the state ACCEPT (line 10). It t is not reachable from s, this path rejects (line 12). □

3.2 FewUL algorithm for Reach on *min-poly* graphs

In this section, we introduce a modification of the previous algorithm to work for *min-poly* graphs. Guessing a constant number of minimum-length paths simultaneously was possible in $\mathcal{O}(\log n)$ space. However, to do so for polynomially many paths would require significantly more space. So, instead of storing entire paths, we will have to assign each path a unique value, and then guess them one by one in some order corresponding to the assigned values. The following is a description of how this is implemented.

Ordering of Paths: For any ℓ-length path $\pi : s \rightsquigarrow v$, we define $\pi(i)$ as the label of the i^{th} vertex of the path, and $\phi(\pi) = \sum_{i=0}^{\ell} 2^{ni+\pi(i)}$. The bit representation of $\phi(\pi)$ is $n \times (\ell+1)$-bits long and if we divide it into $(\ell+1)$ packets of n-bits each, the i^{th} packet has only the bit corresponding to the position $\pi(i)$ set to 1 and all the other bits set to 0. It is easy to see that $\phi(\pi)$ is unique for each path π.

Now, $\phi(\pi)$ is unique for π, but it takes polynomially many bits for representation. So, we need a $\mathcal{O}(\log n)$-bit integer, i.e. an integer which is polynomial in n, and which for each of the vertex $v \in V$, hashes perfectly, the $\phi(\pi)$ values of all the minimum-length $s \rightsquigarrow v$ paths.

We define another term, $\phi_m(\pi) = \phi(\pi) \mod m$. We say that any integer m is *good* for a vertex $v \in V$, if no two minimum-length $s \rightsquigarrow v$ paths p_1 and p_2 satisfy $\phi_m(p_1) = \phi_m(p_2)$. We say that m is *good* for the graph G, if it is *good* for all $v \in V$. We say that m is *k-good* for G, if it is *good* for all the vertices in C_k. The following proposition ensures that there is always a polynomial sized good m.

Proposition 3.1. *[6] For every constant c there is a constant c' so that for every set S of n^2-bit integers with $|S| \leq n^c$ there is a $c' \log n$-bit prime number m so that for all $x, y \in S$, $x \neq y \implies x \not\equiv y \mod m$.*

Guessing Paths in Lexicographic Order: Our algorithm will require guessing paths to a vertex v and checking if the guessed paths are in lexicographic order w.r.t. ϕ_m. Here, we outline a method of doing this in log-space.

Let c be a counter of $\log \ell$ bits to keep track of how far we have traversed along a path. Initialize this to 0. Assign $\log n$ bits to store the current vertex ρ of the current path π. Let π' be the previous

path. Assign two variables, δ_π and $\delta_{\pi'}$, of $\log m$ bits each. to store the intermediate value of $\phi_m(\pi)$ and previously calculated final value of $\phi_m(\pi')$ respectively. Repeat the following two steps until c reaches ℓ.

- $\delta_\pi = (\delta_\pi + 2^{nc+\rho}) \mod m$.
- Increment c and choose one of ρ's neighbour vertices non-deterministically and replace ρ by this neighbour.

Setting δ_π to $\delta_{\pi'}$ and setting δ_π, ρ and c to 0, repeat the steps in the previous paragraph till we have guessed all the q paths. Each time, before updating $\delta_{\pi'}$, check if δ_π is strictly less than $\delta_{\pi'}$. If not, reject there itself.

(Note: To calculate $2^{cn} \mod m$ in log-space, we first initialize $\log n$ bits to 0 and we keep a counter which goes from 0 to cn and each time we increment it, we shift the current value in the $\log n$ bits to the left by 1 bit to double it and if it exceeds m, we subtract m from it.)

Idea: Here, if we have a *good* m, the algorithm runs as in the case of *min-const* graphs, except a slight change in the step where we guess the minimum-length paths for any vertex x. But, we are still faced with the problem of obtaining a *good* m. In the following set of routines, we will guess the value of m and use it while simultaneously verifying if it is indeed *good*. Note that this routine will not be unambiguous any longer, because there could be several choices of m which are *good* for the given graph. However, each choice of m will lead to exactly one accept state. Hence, we can label these accept configurations with their respective choices of m, thus making it a FewUL routine.

Algorithm: The following is a FewUL algorithm which accepts the language:

$$\text{Min-Poly Reach}_c = \{(G(V,E), s, t) \mid \exists\ s \rightsquigarrow t \text{ path and } \forall v \in V, p(v) \leq n^c\}$$

Algorithm 3.4 Main-min-poly-few: Main FewUL routine for Min-Poly Reach$_c$

1: **Input**: (G, s, t)
2: Non-deterministically guess $2 \leq m < n^{c'}$
3: $k := 1$
4: $c_0 := 1; \Sigma_0 := 0; p_0 := 1$
5: $(c_1, \Sigma_1, p_1) =$ Update-min-poly$(G, s, 0, c_0, \Sigma_0, p_0, m)$
6: **while** $k < n-1$ and $(c_{k-1}, \Sigma_{k-1}, p_{k-1}) \neq (c_k, \Sigma_k, p_k)$ **do**
7: $\quad (c_{k+1}, \Sigma_{k+1}, p_{k+1}) =$ Update-min-poly$(G, s, k, c_k, \Sigma_k, p_k, m)$
8: $\quad k := k + 1$
9: **end while**
10: **if** Check-min-poly$(G, s, t, k, c_k, \Sigma_k, p_k, m) > 0$ **then**
11: $\quad$ Go to ACCEPT-m
12: **else**
13: $\quad$ REJECT
14: **end if**

Algorithm 3.5 UPDATE-MIN-POLY: Deterministic (barring CHECK-MIN-POLY calls) routine computing c_{k+1}, Σ_{k+1} and p_{k+1}

Input: $(G, s, k, c_k, \Sigma_k, p_k, m)$
Output: $c_{k+1}, \Sigma_{k+1}, p_{k+1}$
$c_{k+1} := c_k$; $\Sigma_{k+1} := \Sigma_k$; $p_{k+1} := p_k$;
$num := 0$;
for $v \in V$ **do**
  if CHECK-MIN-POLY$(G, s, v, k, c_k, \Sigma_k, p_k, m) = 0$ **then**
    for x such that $(x, v) \in E$ **do**
      $num := num +$ CHECK-MIN-POLY$(G, s, x, k, c_k, \Sigma_k, p_k, m)$;
      if $num > n^c$ **then**
        REJECT
      end if
    end for
    if $num > 0$ **then**
      $c_{k+1} := c_{k+1} + 1; \Sigma_{k+1} := \Sigma_{k+1} + k + 1; p_{k+1} := p_{k+1} + num$;
    end if
  end if
end for

Algorithm 3.6 CHECK-MIN-POLY: UL routine to compute $p(v)$, if $d(v) \leq k$ (returns 0 if $d(v) > k$)

Input: $(G, s, v, k, c_k, \Sigma_k, p_k, m)$
$count := 0; sum := 0; paths := 0; paths.to.v := 0$;
for $x \in V$ **do**
  Nondeterministically guess $0 \leq q \leq n^c$
  if $q \neq 0$ **then**
    Nondeterministically guess $0 \leq \ell \leq k$
    Nondeterministically guess q paths $p_1, p_2, \ldots p_q$ of length exactly ℓ each from s to x.
    if ($(\exists i < j, \phi_m(p_i) \leq \phi_m(p_j))$ OR (paths are not valid)) **then**
      REJECT
    end if
    $count := count + 1; sum := sum + \ell; paths := paths + q$;
    if $x = v$ **then**
      $paths.to.v := q$;
    end if
  end if
end for
if $count = c_k$, $sum = \Sigma_k$ and $paths = p_k$ **then**
  Return the value of $paths.to.v$
else
  REJECT
end if

Proof of Correctness

Claim 3.2.1. *If m is* k-good *for G, and G is* min-poly *considering only vertices in C_k, given the correct values of c_k, Σ_k and p_k, the algorithm* Check-min-poly *has exactly one non-rejecting path, and it returns the correct value of $p(v)$ if $d(v) \leq k$ and 0 if $d(v) > k$.*

Proof. We argue that, since m is *k-good* for G, $\forall x \in V$, there is a unique way to guess the values of q and ℓ, such that the equalities $count = c_k, sum = \Sigma_k$ and $paths = p_k$ are satisfied. The only difference between the routines Check-min-const and Check-min-poly is, the way of guessing q many paths of length ℓ. But, we have seen above that, if m is *k-good* for G, the guessing procedure in Check-min-poly will also be unambiguous, as all the vertices for which minimum-length paths are being guessed are in C_k. So, the proof of Claim 3.1.1 works for this claim too. □

Claim 3.2.2. *If m is* k-good *for G, G is* min-poly *considering only vertices in C_k and correct values of c_k, Σ_k and p_k are given, the algorithm* Update-min-poly *runs deterministically barring the calls to the routine* Check-min-poly*, and computes the correct values of c_{k+1}, Σ_{k+1} and p_{k+1}. It also rejects if G is not* min-poly *considering only vertices in C_{k+1}) (since now the check at line 9 is changed to $num > n^c$).*

Proof. The routine Update-min-poly runs similar to Update-min-const. The only difference is that, it makes calls to Check-min-poly instead to Check-min-const. In Claim 3.2.1, we have seen that if m is *k-good* for G, Check-min-poly works same as Check-min-const, and is correct. Hence, by the same proof as that of Claim 3.1.2, this claim too is correct. □

Observation 3.2. *Observe that, since G is* min-poly *considering only vertices in C_0 and we begin with the correct values of c_0, Σ_0 and p_0, by induction, Claims 3.2.1 and 3.2.2 imply that the values of c_k, Σ_k and p_k calculated after every iteration of while-loop (lines 6 to 9) in the* Main-min-poly-few *algorithm are correct, if m is* good *for G. Also, the iteration ends without rejecting, if and only if the graph G is* min-poly *upto level-k.*

Claim 3.2.3. *If m is* good *for G, the algorithm* Main-min-poly-few *has at most one path to the state ACCEPT-m.*

Proof. Using Observation 3.2 and Claim 3.2.1, we know that there is exactly one non-rejecting path in each call to Check-min-poly. Thus, there is exactly one non-rejecting path in each call to Update-min-poly, as Update-min-poly is deterministic barring the calls to Check-min-poly. Similarly, there is exactly one non-rejecting path in Main-min-poly-few, as Main-min-poly-few for a particular choice of m, is deterministic barring the calls to Update-min-poly. If t is indeed reachable from s, this non-rejecting path goes to ACCEPT-m, as m is guessed initially and is not changed thereafter. □

Claim 3.2.4. *If m is not* good *for G, irrespective of whether G is* min-poly *or not,* Main-min-poly-few *will always reject.*

Proof. Let k be the smallest integer such that m is not *k-good* for G (i.e. it is *(k-1)-good* for G), and let v be the lexiographically first vertex, such that there exist atleast two minimum-length $s \rightsquigarrow v$ paths p_1 and p_2 satisfying $\phi_m(p_1) = \phi_m(p_2)$.

If G is *min-poly*, the first k calls to Update-min-poly runs unambiguously to give correct output. In the $(k+1)^{th}$ call to Update-min-poly (we are having the correct values of c_k, Σ_k and p_k and the fourth parameter passed to any Check-min-poly call will be k), consider the first call to

CHECK-MIN-POLY. For $x = v$, if we guess $q = p(v)$, then the paths cannot be in strictly decreasing order w.r.t. ϕ_m and the algorithm will reject. If we guess $q > p(v)$, then the algorithm will fail to find q paths and reject. If we guess $q < p(v)$, then *paths* will never be equal to p_k, as the q for some other vertex u will then need to be greater than $p(u)$ (for *paths* to become equal to p_k), which is not possible.

If G is not *min-poly*, then there is a smallest integer k' such that G is *min-poly* upto level-$(k'-1)$ and is not *min-poly* upto level-k'. If $k' \leq k$ then by Claim 3.2.2 first $k'-1$ calls to UPDATE-MIN-POLY runs to give correct output and the k'^{th} call to UPDATE-MIN-POLY rejects. The case $k' > k$ has the same proof as the case when G is *min-poly*. □

Lemma 3.2. *The algorithm* MAIN-MIN-POLY-FEW *is correct and* FewUL.

Proof. If m is not *good* for G, then the algorithm MAIN-MIN-POLY-FEW will reject according to the Claim 3.2.4. If m is *good* for G, then there is at most one path which reaches ACCEPT-m (Claim 3.2.3). As there are polynomially many possible values of m, MAIN-MIN-POLY-FEW is FewUL. After covering all the reachable vertices, the while loop (lines 6 to 9) in MAIN-MIN-POLY-FEW terminates with correct values of c_k, Σ_k and p_k, if m is *good* for G and G is *min-poly* (Observation 3.2). Before reaching ACCEPT-m we do a final check to see whether or not vertex t has been covered. As this case occurs only when m is *good* for G and G is *min-poly* (Claims 3.2.3 and 3.2.4), the correct value of $p(t)$ will be returned (Claim 3.2.1) and thus the final decision will be correct. □

3.3 UL algorithm for REACH on *min-poly* graphs

The algorithm presented in the previous section is not unambiguous because there can be more than one *good* m. To address this, we modify the MAIN-MIN-POLY-FEW routine in such a way that we always use the least *good* m (let us call this integer f). The CHECK-MIN-POLY and UPDATE-MIN-POLY routines are already unambiguous and need no change.

Idea: The main idea is to non-deterministically guess f, and to verify that f is the smallest *good* integer for the graph G. This is done using an unambiguous algorithm FIND-FAULT-MIN, which given an integer, verifies that it is not *good*. This routine is run over all the values less than f. Finally when we reach f, we run an algorithm similar to MAIN-MIN-POLY-FEW and accept if and only if it is *good* and there is a path from s to t.

The challenge, in this case, is to make sure that FIND-FAULT-MIN also runs unambiguously. This is achieved as follows. If an integer $m < f$ is not *good*, then there must be a least integer $k_1(m)$ such that m is not $k_1(m)$-*good*. The algorithm FIND-FAULT-MIN guesses k_1 and tries to check if $k_1 = k_1(m)$. Till one level before k_1, it repeatedly calls the algorithm UPDATE-MIN-POLY to update c_k, Σ_k and p_k values. Then, it stops updating these values and tries to find a vertex v with $d(v) = k_1$ in order to certify that m is not k_1-*good*. For any such vertex v, there must exist $a, b \in V$ such that a, b are in-neighbours of v at distance $k_1 - 1$ from s and there must be two paths, p_a through a and p_b through b such that $\phi_m(p_a) = \phi_m(p_b)$.

A UL routine FIND-MATCH$((G, s, k, a, b, \alpha, \beta, m)$ guesses α (respectively β) number of $s \rightsquigarrow a$ ($s \rightsquigarrow b$) paths and pairwise checks for collision w.r.t. the function ϕ_m between $s \rightsquigarrow a$ and $s \rightsquigarrow b$ paths.

Algorithm: The following is a UL algorithm which decides the language MIN-POLY REACH$_c$.

Algorithm 3.7 MAIN-MIN-POLY: Main UL routine for MIN-POLY REACH$_c$

Input: (G, s, t)
Non-deterministically guess $2 \leq f < n^{c'}$[4]
$m := 2$
while $m < f$ **do**
  FIND-FAULT-MIN(G, s, m)
  $m := m + 1$
end while
$k := 1$
$c_0 := 1; \Sigma_0 := 0; p_0 := 1$
$(c_1, \Sigma_1, p_1) =$ UPDATE-MIN-POLY$(G, s, 0, c_0, \Sigma_0, p_0, m)$
while $k < n - 1$ and $(c_{k-1}, \Sigma_{k-1}, p_{k-1}) \neq (c_k, \Sigma_k, p_k)$ **do**
  $(c_{k+1}, \Sigma_{k+1}, p_{k+1}) :=$ UPDATE-MIN-POLY$(G, s, k, c_k, \Sigma_k, p_k, m)$
  $k := k + 1$
end while
if CHECK-MIN-POLY$(G, s, t, k, c_k, \Sigma_k, p_k, m) > 0$ **then**
  ACCEPT
else
  REJECT
end if

Algorithm 3.8 FIND-FAULT-MIN: UL routine to check if m is not *good* for G

Input: (G, s, m)
non-deterministically guess $1 < k_1 < n$
$c_0 := 1; \Sigma_0 := 0; p_0 := 1; k := 1$
while $k < k_1$ **do**
  $(c_k, \Sigma_k, p_k) =$ UPDATE-MIN-POLY$(G, s, k, c_{k-1}, \Sigma_{k-1}, p_{k-1}, m)$
  $k := k + 1$
end while
for $v \in V$ **do**
  if CHECK-MIN-POLY$(G, s, v, k-1, c_{k-1}, \Sigma_{k-1}, p_{k-1}, m) = 0$ **then**
    $valid := false$
    for x such that $(x, v) \in E$ **do**
      if CHECK-MIN-POLY$(G, s, x, k-1, c_{k-1}, \Sigma_{k-1}, p_{k-1}, m) > 0$ **then**
        $valid := true$
      end if
    end for
    if $valid$ **then**
      for $(a, b)|(a, v)$ and $(b, v) \in E$ **do**
        $\alpha :=$ CHECK-MIN-POLY$(G, s, a, k-1, c_{k-1}, \Sigma_{k-1}, p_{k-1}, m)$
        $\beta :=$ CHECK-MIN-POLY$(G, s, b, k-1, c_{k-1}, \Sigma_{k-1}, p_{k-1}, m)$
        if $(\alpha > 0) \wedge (\beta > 0) \wedge$ FIND-MATCH$(G, s, k, a, b, \alpha, \beta, m)$ **then**
          RETURN
        end if
      end for
    end if
  end if
end for
REJECT

Algorithm 3.9 (FIND-MATCH): UL routine to find paths with matching ϕ_m values.

1: **Input**: $(G, s, k, a, b, \alpha, \beta, m)$
2: $collision\_found := false$
3: **for** $i = 1$ to α **do**
4: Guess a path π of length $k-1$ from s to a
5: **if** $(i \geq 2) \wedge (\phi_m(\pi) \geq X)$ **then**
6: REJECT
7: **end if**
8: $X := \phi_m(\pi)$
9: **for** $j = 1$ to β **do**
10: Guess a path π' of length $k-1$ from s to b
11: **if** $(j \geq 2) \wedge (\phi_m(\pi') \geq Y)$ **then**
12: REJECT
13: **end if**
14: $Y := \phi_m(\pi)$
15: **if** $X = Y$ **then**
16: $collision\_found := true$
17: **end if**
18: **end for**
19: **end for**
20: **if** $collision\_found$ **then**
21: Return $true$
22: **else**
23: Return $false$
24: **end if**

Proof of Correctness Let f' be the smallest *good* value for the graph G.

Claim 3.3.1. *If m is not* good *and G is* min-poly *then there exists exactly one non-reject path in* FIND-FAULT-MIN.

Proof. We prove this by considering the following cases :

- If $k_1 > k_1(m) + 1$, then in the while loop (lines 4- 7), when $k = k_1(m) + 1$, UPDATE-MIN-POLY will reject (in the proof of Claim 3.2.4 see the case when G is *min-poly*).
- If $k_1 = k_1(m) + 1$, then in the last iteration of while loop (lines 4- 7) when $k = k_1 - 1 = k_1(m)$, correct values will be computed by UPDATE-MIN-POLY as m is $(k-1)$-*good* and G is *min-poly* (Claim 3.2.2). But, after the loop termination, in line 9 when CHECK-MIN-POLY is called for the first time with fourth parameter as $k - 1 = k_1(m)$, it will reject since m is not $k_1(m)$-*good* (in the proof of Claim 3.2.4 see the case when G is *min-poly*).
- If $k_1 < k_1(m)$, then FIND-MATCH will never find two paths p_1 and p_2 satisfying $\phi_m(p_1) = \phi_m(p_2)$, and the if condition on line 15 will never be satisfied (as X will never be equal to Y). So, it will always return $false$ and thus, FIND-FAULT-MIN will reject at line 27.
- If $k_1 = k_1(m)$, the values c_k, Σ_k and p_k will be updated by UPDATE-MIN-POLY calls in the while loop (lines 4-7), until k becomes equal to k_1 (or $k_1(m)$). Let u be the lexicographically

first vertex such that there exist two $s \rightsquigarrow u$ paths p_1 and p_2 satisfying $\phi_m(p_1) = \phi_m(p_2)$. When $v = u$, in the call to FIND-MATCH at line 20, the if condition on line 15 will be satisfied when $X = \phi_m(p_1)$ and $Y = \phi_m(p_2)$, setting the variable $collision\_found$ to be *true*, and in line 21 *true* will be returned as a final boolean value. So, in line 21 FIND-FAULT-MIN will return to MAIN-MIN-POLY, and this is the only non-reject path.

□

Claim 3.3.2. *If m is* good *and G is* min-poly*, then* FIND-FAULT-MIN *rejects.*

Proof. Notice that, irrespective of the value of k_1 guessed, FIND-MATCH will not be able to find two paths p_1 and p_2 such that $\phi_m(p_1) = \phi_m(p_2)$ as m is *good*. Hence, in line 21, FIND-FAULT-MIN algorithm will never return and thus will reject in line 27. □

Theorem 3. *Testing reachability in graphs augmented with log-space computable* min-poly *weighting schemes, can be done by a non-deterministic log-space algorithm unambiguously and hence is in the complexity class* UL.

Proof. To prove this, we only need to show that the algorithm MAIN-MIN-POLY is correct and UL.

If G is not *min-poly*, MAIN-MIN-POLY will always reject irrespective of what happens in the while loop (lines 4-7), because from line 8 onwards it runs similar to MAIN-MIN-POLY-FEW with $m = f$. Irrespective of m MAIN-MIN-POLY-FEW rejects when the input graph is not *min-poly* (Theorem 3.2), and hence MAIN-MIN-POLY will also reject.

Now, consider the case when G is *min-poly*. We argue that if $f = f'$, MAIN-MIN-POLY accepts in at most one path, and if $f \neq f'$, MAIN-MIN-POLY rejects. Consider the case $f = f'$. In each iteration of the first while loop (lines 4-7) in MAIN-MIN-POLY, m is not *good* and thus by claim 3.3.1, the while loop terminates in exactly one path. The rest of the algorithm (lines 8-19) is identical to MAIN-MIN-POLY-FEW. So, by the proof of Theorem 3.2 there is at most one accept path. Note, that here unlike in MAIN-MIN-POLY-FEW, we will reach a unique accept state corresponding to $m = f = f'$.

Now consider $f \neq f'$. If $f < f'$, then at line 7, when the first while loop terminates, $m = f < f'$, the rest of the algorithm will reject as f is not *good* for G (Claim 3.2.4). If $f > f'$, then when in first while loop $m = f'$, FIND-FAULT-MIN will reject, as m is *good* (Claim 3.3.2).

Now we argue correctness. As argued, line 15 in MAIN-MIN-POLY will be reached only when $f = f'$. At this point, c_k, Σ_k and p_k are calculated correctly, as Observation 3.2 still holds. Thus, by Claim 3.2.1, CHECK-MIN-POLY outputs the correct value of $p(t)$ as $m = f'$ is *good* and thus the final result is correct. □

4 Reachability on *max-poly* layered DAGs

It is known that REACH on general directed graphs reduces to REACH on layered DAGs, making the latter also NL-complete. In this section, we will look to construct a UL algorithm to solve REACH on layered DAGs, rather than general graphs. [5]

[5] In this connection, we remark that it is unclear how REACH in the case of arbitrary DAGs can be reduced in log-space to REACH in layered DAGs while preserving the *max-poly* property. It is easy to verify that the standard reduction fails in this aspect. The reduction works as follows : let $(G = (V, E), s, t)$ be an instance of the original reachability problem. Define G' we make $|V|$ labelled copies of V. Adjacency between adjacent copies of V is made

In order to arrive at the algorithm for REACH in *max-poly* layered DAGs, we solve a harder problem on a more specific class of graphs. This is a variant of the LONGPATH problem (Given (G, s, t, j) where s and t are vertices in the graph G, and j is an integer - the LONGPATH problem asks to check if there is a path from s to t of length at least j) where the graph G has a unique source s.

We first give the reduction from REACH on layered DAGs to LONGPATH on single-source DAGs.

4.1 Reducing REACH on layered DAGs to LONGPATH on single-source DAGs

This section presents a reduction from REACH on layered DAGs to LONGPATH on single-source DAGs. Since the former is known to be complete for the class NL, it makes LONGPATH on single-source DAGs NL-complete as well.

Lemma 4.1. REACH *on layered DAGs reduces via a log-space many-one reduction to* LONGPATH *on single-source DAGs.*

More formally, there is a function f, computable in log-space, that transforms an instance $(G(V, E), s, t)$ of REACH *to an instance $(G'(V', E'), s', t, 2n + 1)$ of* LONGPATH*, where $n = |V|$, such that t is reachable from s in G if and only if there exists a path of length at least $2n + 1$ from s' to t in G'. In addition, if G is* max-unique *(*max-poly*), then G' is* max-unique *(*max-poly*).*

Proof. Since the graph $G(V, E)$ is a layered DAG, we can assume that the vertices of the G are numbered such that, edges always go from a lower numbered vertex to a higher numbered vertex. Let $V = \{v_1, v_2, \ldots, v_n\}$ be this numbering. We will construct $G'(V', E')$ as follows: In addition to the edges among the vertices in V, we add a new source vertex s' and add edges from s' to all other vertices in V.

Formally,

$$V' = V \cup \{s'\}$$

$$E' = E \cup \{(s', v) \mid v \in V\}$$

Further, we 'apply' the following weighting scheme to G':

$$W(u, v) = \begin{cases} 2n, & \text{if } (u, v) = (s', s). \\ 2i, & \text{if } (u, v) = (s', v_i) \ \forall\ 1 \leq i \leq n. \\ 1, & \text{otherwise.} \end{cases} \tag{1}$$

The weight of the edge $(s', s) = 2n$ and for vertices $v_i \neq s$, weight of (s', v_i) is $2i$. Note that G' has exactly one source vertex s' and hence is a valid input for our algorithm to solve LONGPATH.

Now we argue the that if G had a unique path (polynomially many paths) of maximum length from s to any vertex v, then so will be the case with G'. This condition is easily seen for $v \notin V$. For a vertex $v_i \in V$, we claim that among all the paths not going through s, there is exactly one path of maximum length and this is the path corresponding to the edge (s', v_i) of length $2i$. If not, choose a longest path (say p) which is not corresponding to the edge (s', v_i). Let v_j $(j < i)$ be the first vertex in p from V. Clearly, p must use the path corresponding to the weighted edge (s', v_j). Hence, the length of the path p can at most be $2j + (i - j) = i + j < 2i$. This contradicts the choice of p.

same as the adjacency of the graph G. Connect adjacent copies of t. A layering is automatically obtained as a part of the reduction.

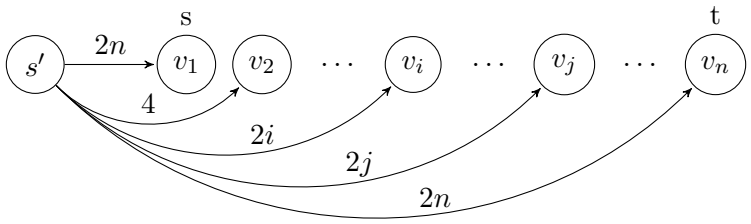

Figure 1: Construction for the reduction described in Lemma 4.1

Thus, for a vertex $v_i \in V$ that is not reachable from s, the maximum length path in G' is unique. For a vertex $v_i \in V$ that is reachable from s, the maximum length path not through s is of weight exactly $2i$, but the paths from s' to v_i through s are of length at least $2n + 1 > 2i$. Additionally, we can see that, if there were ℓ paths of maximum length from s to any vertex v_i in G, then the number of maximum length paths from s' to v_i is also ℓ.

We now argue correctness of our reduction. Suppose that t is not reachable from s in G. In this case, none of the paths from s' to t will pass through s. Hence, using the above argument, we know that the length of any path from s' to t cannot be greater than $2n$. On the other hand, if t is reachable from s in G (say by path p), then the path (s', s) concatenated with p is a path of length $\geq 2n + 1$ from s' to t. □

Observation 4.1. *Since G is a layered DAG, by labelling s' with an integer less than v_0, we can make sure that G' is also a DAG with all the edges moving from vertices with lower labels to those with higher labels.*

Now we turn to this special case of LongPath problem. LongPath for *max-unique* DAGs with a unique source has been studied by [9]. The UL algorithm in [9] is for LongPath on *max-unique* graphs having a single sink t. In our version of LongPath, we will consider paths from s (as opposed to paths to t in [9]) and hence we will consider only graphs with a unique source s. We will extend their algorithm to *max-poly* layered DAGs, by first giving a FewUL algorithm, and then converting it to a UL algorithm using a strategy similar to the *min-poly* Reach algorithm in Section 3.

4.2 FewUL algorithm for LongPath on *max-poly* single-source DAGs

In a way similar to our adaptation of algorithm for *min-unique* graphs of [12] to work with *min-poly* graphs, we adapt the algorithm proposed in [9] for *max-unique* DAGs (with a unique sink) to the case for *max-poly* DAGs with a unique source. Along with the reduction we mentioned above from Reach to LongPath in such graphs (preserving their *max-unique* or *max-poly* property), this gives an algorithm for reachability testing in such graphs.

We introduce notation required for our exposition. $D(v)$ is the length of the longest $s \rightsquigarrow v$ path in G. Additionally, we define $P(v)$ as the number of maximum-length $s \rightsquigarrow v$ paths in G. We reuse the term C_k to denote the set $\{v \mid D(v) < k\}$. We also reuse the following - $c_k = |C_k|$, $\Sigma_k = \sum_{v \in C_k} D(v)$ and $p_k = \sum_{v \in C_k} P(v)$.

We build the intuition through an example setting where an idea similar to that of the algorithm CHECK-MIN-POLY fails. Suppose we have the correct values of c_k, Σ_k and p_k. Even then, suppose for a vertex v, we guess $D(v)$[6] $< k$ whereas originally $D(v) \geq k$. The algorithm, in this non-deterministic choice can still make it to the original summation by guessing for another u that $D(u) \geq k$ whereas originally $D(u) < k$. Since the algorithm is not verifying guesses of the kind $D(u) \geq k$ (that is, $q = 0$). In [9], this problem is addressed by introducing a new parameter $T = \sum_{v \in V} D(v)$. The value of M is also non-deterministically guessed, which if guessed correctly, will facilitate verification of guess $D(u) \geq k$.

In our modification, along with T, we use another parameter $S = \sum_{v \in V} P(v)$. Both T and S are initially guessed (guessed values are called M and P respectively) and verified in the end.

Idea: The routine CHECK-MAX-POLY$(G, s, v, c_k, \Sigma_k, p_k, m)$, which given the correct values of c_k, Σ_k and p_k, tests whether $D(v) \geq k$ unambiguously and outputs $(D(v), P(v))$ if $D(v) < k$ and outputs $(0, 0)$ if $D(v) \geq k$.

For each vertex $x \in V$, the routine guesses whether or not $D(x) \geq k$. It also guesses the number of longest $s \rightsquigarrow x$ paths and sequentially guesses each of these paths. The algorithm keeps a count of vertices in C_k. Additionally, both for vertices in C_k and those outside, the routine separately keeps an account of the number and length of longest paths from s to each vertex. It maintains the sums of these values and later verifies these values with the known values of c_k, Σ_k, p_k, M and P. These checks will all hold true only at the unique path where all the values are guessed correctly. This makes sure that the routine has at most one non-reject computational path. The routine returns $(0, 0)$ if it had guessed $D(v) \geq k$, and returns $(D(v), P(v))$ otherwise.

As for the inductive computation of c_{k+1}, Σ_{k+1} and p_{k+1} from c_k, Σ_k, and p_k, UPDATE-MAX-POLY finds every vertex v such that $D(v)$ is exactly equal to k. To do this, the routine makes sure that $D(v) \geq k$ and does not have any in-neighbour x with $D(x) \geq k$. The routine then updates the values of c_{k+1}, Σ_{k+1} and p_{k+1} with the values $D(v)$ and $P(v)$ obtained from every such v. The routine also checks if the *max-poly* condition is followed, and rejects if not.

The main reachability test algorithm, given (G', s', t') as the input, constructs, in log-space, the instance (G, s, t, j) of the special case of LONGPATH problem. It runs the remaining algorithm with this new graph. The algorithm guesses m, M and P, and using these values, inductively computes the values of c_k, Σ_k and p_k until all vertices are covered. Finally, it checks whether or not $D(t) \geq j$, and returns the answer.

Algorithm: The following is a FewUL algorithm which decides the language:

$$\text{MAX-POLY LONGPATH}_c = \{(G(V, E), s, t, j) \mid \exists s \rightsquigarrow t \text{ path of length } \geq j \text{ and } \forall v \in V, P(v) \leq n^c\}$$

[6] D(v)?

Algorithm 4.1 CHECK-MAX-POLY: An unambiguous routine to determine if $D(v) \geq k$ and return the values $(D(v), P(v))$ if not.

Input: $(G, s, v, k, c_k, \Sigma_k, p_k, m, f)$
$count := 0; sum := 0; sum' := 0; paths := 0; paths' := 0; paths.to.v := 0;$ $length.to.v := 0;$
for $x \in V$ **do**
  Non-deterministically guess if $D(x) \geq k$
  if the guess is NO **then**
    Non-deterministically guess $0 \leq \ell < k$ and $0 < q \leq n^c$
    Non-deterministically guess q paths $p_1, p_2, \ldots p_q$ of length ℓ each from s to x.
    if ($(\exists i < j, \phi_f(p_i) \leq \phi_f(p_j))$ OR (paths are not valid)) **then**
      REJECT
    end if
    Non-deterministically guess q paths $p_1, p_2, \ldots p_q$ of length ℓ each from s to x.
    if (($(\exists i < j, \phi_m(p_i) \leq \phi_m(p_j))$ OR (paths are not valid)) **then**
      REJECT
    end if
    $count := count + 1; sum := sum + \ell; paths := paths + q;$
    if $x = v$ **then**
      $paths.to.v := q;$
      $length.to.v := \ell;$
    end if
  else
    Non-deterministically guess $0 < q \leq n^c$
    Non-deterministically guess $k \leq \ell < n$
    Non-deterministically guess q paths $p_1, p_2, \ldots p_q$ of length ℓ each from s to x.
    if ($(\exists i < j, \phi_f(p_i) \leq \phi_f(p_j))$ OR (paths are not valid)) **then**
      REJECT
    end if
    $sum' := sum' + \ell; paths' := paths' + q;$
  end if
end for
if $count = c_k$; $sum = \Sigma_k$; $paths = p_k$; $paths' + paths = P$; $sum' + sum = M$ **then**
  Return the value of $paths.to.v$ and $length.to.v$.
else
  REJECT
end if

Algorithm 4.2 UPDATE-MAX-POLY: An unambiguous routine computing c_{k+1}, Σ_{k+1} and p_{k+1} from c_k, Σ_k and p_k.

Input: $(G, s, k, c_k, \Sigma_k, p_k, m, f)$
$c_{k+1} := c_k$; $\Sigma_{k+1} := \Sigma_k$; $p_{k+1} := p_k$; $flag := true$; $num := 0$;
for $v \in V$ **do**
  if CHECK-MAX-POLY$(G, s, v, k, c_k, \Sigma_k, p_k, m, m)$ **then**
    for x such that $(x, v) \in E$ **do**
      if CHECK-MAX-POLY$(G, s, x, k, c_k, \Sigma_k, p_k, m, m)$ **then**
        $flag := false$
      end if
      if $D(x) = k - 1$ **then**
        $num := num + P(x)$;
      end if
      if $num > n^c$ **then**
        REJECT
      end if
    end for
    if $flag$ **then**
      $c_{k+1} := c_{k+1} + 1; \Sigma_{k+1} := \Sigma_{k+1} + k; p_{k+1} := p_{k+1} + num$;
    end if
  end if
end for

Algorithm 4.3 MAIN-MAX-POLY-FEW: The main FewUL algorithm for LONGPATH

Input: (G, s, t)
Non-deterministically guess $2 < m < n^c$
Non-deterministically guess $0 < M < n^2$ $(M = \sum_{v \in V} D(v))$
Non-deterministically guess $1 < P < n^{c+1}$ $(P = \sum_{v \in V} P(v))$
$k := 0$;
$c_0 := 0; \Sigma_0 := 0; p_0 := 0$;
$c_1 := 1; \Sigma_1 := 0; p_1 := 1$;
while $c_{k-1} \neq c_k$ **do**
  $k := k + 1$;
  $c_k, \Sigma_k, p_k :=$ UPDATE-MAX-POLY$(G, s, k, c_k, \Sigma_k, p_k, m, m)$
end while
if CHECK-MAX-POLY$(G, s, t, j, c_j, \Sigma_j, p_j, m, m)$ **then**
  REJECT
else
  go to state ACCEPT-m
end if

Proof of correctness

Claim 4.2.1. *Irrespective of the guessed values of M and P, if the input values c_k, Σ_k and p_k are correct, then all non-reject paths of* CHECK-MAX-POLY *return the correct $P(v)$ and $D(v)$ for v such*

that $D(v) < k$. (For other vertices it returns $(0, 0)$)

Proof. This can be analyzed by the following cases :

- Suppose that $D(v) < k$. If the algorithm guesses otherwise, it will be unable to guess a witness path of length $\geq k$, and will hence reject. If it guesses correctly, then,
  - If the algorithm guesses $\ell > D(v)$, then it will not be able to guess any path of length ℓ and hence will reject.
  - If the algorithm guesses $\ell < D(v)$, then the value of *sum* will not match Σ_k unless the algorithm guesses $\ell > D(y)$ for some other vertex y, which will lead it to reject.
  - Now, if the algorithm guesses $q > P(v)$ for some $v \in V$, then it will fail to compute q such paths of length ℓ (because $\ell = D(v)$).
  - If the algorithm guesses $q < P(v)$, then *paths* will not match p_k as, similar to the case with ℓ, compensation is not possible.

  Hence, for each v with $D(v) < k$, the algorithm returns the correct $P(v)$ and $D(v)$.
- Now suppose that $D(v) \geq k$. If the algorithm guesses otherwise, this causes an extra increment for *count*. Thus, the value of *count* will not match c_k, because to do so, the algorithm must guess $D(u) \geq k$ for some vertex u for which this is not true. But, from the argument used in the previous case, we know that this is not possible. Hence, it will reject. However if the algorithm guesses correctly, then the algorithm outputs $P(v) = 0$ and $D(v) = 0$.

□

Claim 4.2.2. *If the algorithm* Check-max-poly *works correctly for parameter k, then given the correct values of c_k, Σ_k and p_k, the algorithm* Update-max-poly *computes the correct values of c_{k+1}, Σ_{k+1} and p_{k+1}.*

Proof. The algorithm first assigns $c_{k+1} := c_k, \Sigma_{k+1} := \Sigma_k$ and $p_{k+1} := p_k$. Now, to update these values we need the exact set of vertices v with $D(v) = k$. For each such v, the algorithm increments c_{k+1} by 1 and Σ_{k+1} by k. The algorithm, for each v, checks whether $D(v) \geq k$ and for each of its neighbours u, checks if $D(u) < k$. If both the conditions are true, then we know that $D(v) = k$. For each such neighbour u for which $D(u) = k - 1$, the algorithm adds $P(u)$ to *num* which at the end of the loop results in $num = P(v)$. The algorithm then increments p_{k+1} by *num*. Thus all the three parameters are updated correctly. □

Observation 4.2. *Since we begin with the correct values of c_0, Σ_0 and p_0, by induction, Claims 4.2.1 and 4.2.2 imply that the values of c_k, Σ_k and p_k calculated at any time in the algorithm are always correct.*

Claim 4.2.3. *If $M = T$, $P = S$ and m is* good*, and if the input values c_k, Σ_k and p_k are correct, then there is exactly one non-reject path in* Check-max-poly

Proof. The equalities $paths' + paths = P$ and $sum' + sum = M$ can only be satisfied if, for each vertex v, the algorithm guesses $\ell = D(v)$ and $q = P(v)$, and all q paths in lexicographic order w.r.t. ϕ_m. This can happen only in a unique way when m is *good*. □

Claim 4.2.4. *If $M = T$, $P = S$ and m is not* good*, given the correct values of c_k, p_k and Σ_k, the algorithm* Check-max-poly *(and hence both the algorithms* Update-max-poly *and* Main-max-poly-few*) always rejects.*

Proof. If m is not *good*, then there exists a vertex v such that the graph has at least two $s \rightsquigarrow v$ paths p_1 and p_2 satisfying $\phi_m(p_1) = \phi_m(p_2)$. So, if $q = P(v)$, then the paths cannot be in strictly decreasing order and the algorithm will reject. If $q > P(v)$, then the algorithm will fail to find q paths and reject. If $q < P(v)$, then *paths* will never be equal to p_k, as the q for some other vertex u will then need to be guessed to be greater than $P(u)$, which will lead it to reject. □

Claim 4.2.5. *If $M \neq T$ or $P \neq S$, then the algorithm* Main-max-poly-few *always rejects irrespective of the guessed value of m.*

Proof. We analyze this by the following cases.

- If $M > T$ ($P > S$), then we will never get paths for which $sum{+}sum' = M$ ($paths{+}paths' = P$) will be true.
- If $M < T$ ($P < S$), and suppose the algorithm Main-max-poly-few reaches the the stage at which $D(v) < k$ for all vertices $v \in V$. At this stage in the algorithm Check-max-poly, we will obtain $sum = \Sigma_k$ ($paths = p_k$) and $sum' = 0$ ($paths' = 0$). However, due to the correctness of the value of Σ_k (p_k), $\Sigma_k = T$ ($p_k = S$). Thus, the check $sum + sum' = M$ ($paths + paths' = S$) will fail.

□

Claim 4.2.6. *If $M = T$, $P = S$ and m is* good*, then there is exactly one path in* Main-max-poly-few *which reaches the state ACCEPT-m.*

Proof. By Claim 4.2.3 and Observation 4.2 we know that there is exactly one non-rejecting path in each call to Check-max-poly. So, there is exactly one non-rejecting path in each call to Update-max-poly, as Update-max-poly is deterministic barring the calls to Check-max-poly. Hence, there is exactly one non-rejecting path in Main-max-poly-few, as Main-max-poly-few (after guessing m, M and P), is deterministic barring the calls to Update-max-poly. This non-rejecting path goes to ACCEPT-m as m is fixed initially and is not changed thereafter. □

Lemma 4.2. *The algorithm* Main-max-poly-few *is correct and is* FewUL*.*

Proof. If the value of m guessed is not *good*, then the algorithm Main-max-poly-few always rejects (Claims 4.2.4 and 4.2.5), and if it is *good*, there is at most one computational path to the state ACCEPT-m (Claims 4.2.5 and 4.2.6). And as the number of choices for m is bounded by a polynomial, Main-max-poly-few is FewUL.

When all the vertices are covered, the while loop in Main-max-poly-few stops. At this point we have correct values of c_k, Σ_k and p_k (Observation 4.2) and before the algorithm reaches ACCEPT-m, it performs a final check to see whether or not $D(t) \geq j$. As this case occurs only when m is *good*, the correct value of $D(t)$ will be returned (Claim 4.2.1) and thus the final decision will be correct. □

4.3 UL algorithm for LongPath on *max-poly* single-source DAGs

The primary issue preventing the algorithm in the previous section from being unambiguous was the non-unique value of m. There is no reason to expect only one integer m to be able to hash the ϕ_m values of the paths in a graph without collisions. To overcome this issue, in the following algorithm, we attempt to find the *least* such m and use this integer to hash the paths in our algorithm.

Idea: The idea is similar to the one used in the case of Reach in *max-poly* graphs. Firstly, we guess this smallest *good* integer f, and verify that all smaller integers are indeed not *good*. However, instead of doing so independently, here we run the algorithm for m and f simultaneously. This is because we want to verify the smaller m values only on the correct path for f. The reason for this is that, due to the added guesses of M and P, the number of non-reject paths on m in the algorithm will not remain unique if the test for m is run independently.

Algorithm: The following is a UL algorithm which decides the language Max-Poly LongPath$_c$.

Proof of correctness

Claim 4.3.1. *If m is not* good *and f is* good *then there exists exactly one non-reject path in* Find-fault-max.

Proof. Since f is *good*, we know that, when $k = n - 1$, in line 33 of Find-fault-max, if M or P are guessed incorrectly, our algorithm will reject (see proof of Claim 4.2.5). So, we will assume that M and P are guessed correctly.

- If $k_1 > k_1(m)$, then in the while loop, when $k = k_1(m)$, Update-max-poly will find two paths p_1 and p_2 such that $\phi_m(p_1) = \phi_m(p_2)$ and will reject.
- If $k_1 < k_1(m)$ then Find-match will never find two paths p_1 and p_2 such that $\phi_m(p_1) = \phi_m(p_2)$ for $k = k_1$. So, it will always return $false$ and thus Find-fault-max will reject at $k = k_1 + 1$.
- If $k_1 = k_1(m)$: let u be the lexicographically first vertex such that there exist two $s \rightsquigarrow u$ paths p_1 and p_2 satisfying $\phi_m(p_1) = \phi_m(p_2)$. Hence, when $v = u$ the algorithm will jump to line 32 and this is the only non-reject path.

□

Claim 4.3.2. *If m is* good *or if f is not* good *then* Find-fault-max *rejects.*

Proof. If f is not *good*, then in the first iteration of the while loop, Update-max-poly will reject due to Claims 4.2.4 and 4.2.5. If m is *good*, then irrespective of k_1 guessed, Find-match will not be able to find two paths p_1 and p_2 such that $\phi_m(p_1) = \phi_m(p_2)$. Hence, the routine Find-fault-max will never break out of the loop in line 23 and will reject in line 31. □

Claim 4.3.3. *If $f = f'$,* Main-max-poly *accepts in at most one path.*

Proof. In case we guess $M \neq T$ or $P \neq S$, the algorithm will reject due to Claim 4.2.5. When $M = T$ and $P = S$, in each iteration of the while loop, $m < f'$ and hence is not *good*. Thus by Claim 4.3.1, all iterations are unambiguous, and the while loop terminates in exactly one path.

Algorithm 4.4 FIND-FAULT-MAX: A UL routine to verify that an integer m is bad for G.

1: **Input**: (G, s, m, f)
2: Non-deterministically guess $1 < k_1 < n$
3: $c_0 := 0; \Sigma_0 := 0; p_0 := 0; k := 1$
4: **while** $k \leq k_1$ **do**
5: Compute c_k, Σ_k and p_k from c_{k-1}, Σ_{k-1} and p_{k-1} using both m and f to hash
6: $k := k + 1$
7: **end while**
8: **for** $v \in V$ **do**
9: **if** $D(v) \geq k_1$ **then**
10: $valid := true$
11: **for** x such that $(x, v) \in E$ **do**
12: **if** $D(x) \geq k_1$ **then**
13: $valid := false$
14: **end if**
15: **end for**
16: **if** $valid$ **then**
17: **for** $(a, b) \mid (a, v)$ and $(b, v) \in E$ **do**
18: **if** $D(a) = k - 1$ **then**
19: **if** $D(b) = k - 1$ **then**
20: $\alpha := P(a)$
21: $\beta := P(b)$
22: **if** $(\alpha > 0) \wedge (\beta > 0) \wedge (\text{FIND-MATCH}(a, b) = true)$ **then**
23: **goto** line 32
24: **end if**
25: **end if**
26: **end if**
27: **end for**
28: **end if**
29: **end if**
30: **end for**
31: REJECT
32: **while** $k < n$ and $(c_{k-1}, \Sigma_{k-1}, p_{k-1}) \neq (c_k, \Sigma_k, p_k)$ **do**
33: Compute c_k, Σ_k and p_k from c_{k-1}, Σ_{k-1} and p_{k-1} using f to hash
34: $k := k + 1$
35: **end while**

Algorithm 4.5 MAIN-MAX-POLY: The main UL routine to solve LONGPATH on *max-poly* single-source DAGs.

1: **Input**: (G, s, t, j)
2: Non-deterministically guess $2 \leq f < n^{c'}$
3: Non-deterministically guess $0 \leq M \leq n^2$ $(M = \sum_{v \in V} D(v))$
4: Non-deterministically guess $1 \leq P \leq n^{c+1}$ $(P = \sum_{v \in V} P(v))$
5: $m := 2$
6: **while** $m < f$ **do**
7: FIND-FAULT-MAX(m)
8: $m := m + 1$
9: **end while**
10: **if** $D(t) \geq j$ **then**
11: ACCEPT
12: **else**
13: REJECT
14: **end if**

The rest of the algorithm is also unambiguous, since CHECK-MAX-POLY is UL (Claim 4.2.3) and Observation 4.2 still holds.

□

Claim 4.3.4. *If $f \neq f'$,* MAIN-MAX-POLY *rejects.*

Proof. If $f < f'$, then f is surely not *good*, and during the first iteration of the while loop, FIND-FAULT-MAX will reject due to Claim 4.3.2. If $f > f'$ then, when $m = f'$ in the while loop, FIND-FAULT-MAX will reject due to Claim 4.3.2 because m is *good*. □

Theorem 4. *Testing reachability in layered DAGs augmented with log-space computable* max-poly *weighting schemes, can be done by a non-deterministic log-space algorithm unambiguously and hence is in the complexity class* UL.

Proof. Lemma 4.1 shows that REACH on layered DAGs can be reduced to LONGPATH on single-source DAGs. Hence, to prove the theorem, we only need to show that the algorithm MAIN-MAX-POLY solves LONGPATH correctly and is in UL.

The unambiguity follows from Claims 4.3.3 and 4.3.4. We will reach line 10 in MAIN-MAX-POLY only when $f = f'$, and at this point, c_k, Σ_k, p_k are calculated correctly, as Observation 4.2 still holds. Thus, by Claim 4.2.2, we get the correct value of $P(t)$ and the final result is correct. □

5 Discussion and Open Problems

We designed UL algorithms for REACH in layered DAGs augmented with *min-poly* or *max-poly* weighting schemes. This is an improvement of previous results by Allender and Reindthart [12], where they designed a UL algorithm for REACH in *min-unique* graphs, and by Limaye, Mahajan and Nimbhorkar [9], where they designed a UL algorithm for LONGPATH in *max-unique* single source DAGs.

UL-equivalence of *min-poly* and *max-poly* weighting schemes: To make the study complete, we also explore the converse of our result - namely deriving weighting schemes from UL algorithms. This has already been studied in the case of min-unique weighting schemes. The following proposition can be argued by a minor variant of a proof in [10]. We include it here for completeness.

Proposition 5.1. *The following statements are equivalent :*

1. NL = UL.
2. *There is a polynomially bounded* UL*-computable min-unique weighting scheme for any DAG.*
3. *There is a polynomially bounded* UL*-computable min-poly weighting scheme for any DAG.*
4. *There is a polynomially bounded* UL*-computable max-unique weighting scheme for any layered DAG.*
5. *There is a polynomially bounded* UL*-computable max-poly weighting scheme for any layered DAG.*

Proof. (2) $\implies$ (3) and (4) $\implies$ (5) follows from the definitions. (1) $\implies$ (2) follows from [10]. (3) $\implies$ (1) follows from Theorem 1. (5) $\implies$ (1) follows from Theorem 2.

To complete the cycle, we will show (1) $\implies$ (4). If NL = UL, we need to provide a UL-computable max-unique weighting scheme for any layered DAG G. We will use a weighting scheme very similar to the one used in Theorem 4.1 of [10]. We include the full proof here for completeness.

The idea (due to [10]) is to compute a spanning tree of G rooted at s using reachability queries. Since NL = co-NL, under the assumption that NL = UL, we can conclude that UL = co-UL. And for any language A in UL, L^A is in UL. Now, consider the language $A = \{(G, s, v, k) \mid$ there is a path from s to v of length $\geq k\}$ is in UL (as it is in NL). L^A is also in UL.

Borrowing the idea from [10], we construct a spanning tree of G rooted at s using queries to A. So, this construction is in UL. For convenience we describe their construction, while at the same time, highlight the difference in assigning weights in our case. A vertex v is said to be in level k if $D(v) = k$. An edge (u, v) is in the tree if for some k, v is in level k and u is the lexicographically first vertex in level $k-1$ which has an edge to v. Now for each edge in the tree, give a weight n^2. For the rest of the edges give a weight 1 (This weighing scheme is opposite to the one used in the original proof). We can see that, of all maximum-length paths to a vertex on level k, there is exactly one with all edges of weight n^2. Hence, the graph is max-unique.

Constructing this tree and checking if an edge (u, v) belongs to this tree can be done trivially in log-space by querying A for u and v. Thus, assigning weights for this graph is in $\mathsf{L}^A \subseteq$ UL. □

We now list down some of the open problems that the work presented in the paper lead to.

Open Problem 1: *Min/Max-poly* weighting scheme design. Are *min-poly* or *max-poly* log-space computable weighting schemes easier to design than *min-unique* log-space computable weighting schemes? In the former case, the restrictions on the number of minimum weight $s \rightsquigarrow v$ paths is less stringent. Hence it is conceivable that such a weighting scheme is easier to design. Designing such a weighting scheme would immediately imply that NL = UL.

Indeed, it may be simpler to design such weighting schemes for restricted graph classes. We know a *min-unique* weighting scheme for grid graphs [5]. If we are able to extend this scheme to make it work for "Monotone 3D Grid Graphs", then NL = UL [5].

Open Problem 2: Reduction between weighting schemes. We have shown that testing reachability in *max-poly* layered DAGs can be done in UL. With respect to layered DAGs, it would be interesting to know if these problems are inter-reducible in log-space. In particular, is there a deterministic log-space algorithm, which given a log-space computable, polynomially bounded, *min-unique* (resp. *min-poly*) weighting scheme for a graph G, produces a *max-unique* (resp. *max-poly*) weighting scheme for G? Notice that, Proposition 5.1 indicates that there is UL reduction of this kind. However, it is not clear if there is a direct deterministic log-space reduction between the two restricted problems (say, in the case of layered DAGs).

Open Problem 3: Can either of the algorithms be extended to a larger class of graphs? We know that extending the algorithm to allow exponentially many minimum/maximum-weight paths is equivalent to solving Reach for general graphs in UL. It will be interesting to see if the algorithms presented in the paper could be extended to an larger class of graphs. We remark that the UL algorithm for *max-poly* graphs work only for layered DAGs, as the layering is critically needed in the reduction to LongPath problem in section 4.1.

References

[1] Eric Allender. Reachability problems: An update. In *Proceedings of Computability in Europe*, pages 25–27, 2007.

[2] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4):675–723, July 2009.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[4] A. Bondy and U.S.R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer London, 2007.

[5] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. *ACM Transations on Computation Theory*, 1(1):4:1–4:17, February 2009.

[6] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM*, 31(3):538–544, June 1984.

[7] Brady Garvin, Derrick Stolee, Raghunath Tewari, and N.V. Vinodchandran. ReachFewL = ReachUL. *Computational Complexity*, 23(1):85–98, 2014.

[8] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17(5):935–938, October 1988.

[9] Nutan Limaye, Meena Mahajan, and Prajakta Nimbhorkar. Longest paths in planar dags in unambiguous logspace. *Chicago Journal of Theoretical Computer Science*, 2010(8). Preliminary version appeared in the Proceedings of Computing: The Australasian Theory Symposium (CATS), pages 101–108, 2009.

[10] Aduri Pavan, Raghunath Tewari, and N. V. Vinodchandran. On the power of unambiguity in log-space. *Computational Complexity*, 21(4):643–670, 2012.

[11] Omer Reingold. Undirected st-connectivity in log-space. In *Proceedings of Symposium of Theory of Computing 2005*, pages 376–385, 2005.

[12] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal of Computing*, 29(4):1118–1131, 2000.

[13] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, November 1988.