

Longest Paths in Planar DAGs in Unambiguous Logspace

Nutan Limaye, Meena Mahajan, Prajakta Nimbhorkar

The Institute of Mathematical Sciences,
Chennai 600 113, India.
Email: {nutan,meena,prajakta}@imsc.res.in

Abstract

Reachability and distance computation are known to be NL-complete in general graphs, but within $UL \cap co-UL$ if the graphs are planar. However, finding longest paths is known to be NP-complete, even for planar graphs. We show that with the combination of planarity and acyclicity, finding the length of the longest path (and also enumerating one such path) is also in $UL \cap co-UL$. The result extends to toroidal DAGs as well. We also address the question of when reachability, distance, and longest path are indeed equivalent on DAGs, and give partial bounds.

When the number of distinct paths is bounded by a polynomial, counting the number of paths is known to be in NL. We show that for planar DAGs with this promise, counting can be done by a UAuxPDA in polynomial time. The UAuxPDA(poly) bound also holds if we want to compute the number of longest paths, or shortest paths, and this number is bounded by a polynomial (irrespective of the total number of paths). Along the way, we show that counting in general DAGs is possible in LogDCFL provided the number of paths is bounded by a polynomial and the target node is the only sink.

Keywords: directed acyclic graphs, reachability, longest path, planar, unambiguous logspace

1 Introduction

Consider the following problems in graphs:

$$\begin{aligned} \text{Reach} &= \left\{ (G, s, t) \mid \begin{array}{l} G \text{ contains a} \\ \text{path from } s \text{ to } t \end{array} \right\} \\ \text{Distance} &= \left\{ (G, s, t, k) \mid \begin{array}{l} G \text{ contains a} \\ \text{path of length} \\ \leq k \text{ from } s \text{ to } t \end{array} \right\} \\ \text{Long-Path} &= \left\{ (G, s, t, k) \mid \begin{array}{l} G \text{ has a simple} \\ \text{path of length} \\ \geq k \text{ from } s \text{ to } t \end{array} \right\} \\ \#\text{Path} &= \left\{ (G, s, t, 1^k) \mid \begin{array}{l} G \text{ has exactly } k \\ \text{paths from } s \text{ to } t \end{array} \right\} \end{aligned}$$

These problems have widely differing complexities: some of the results below are folklore, some are recent advances. Reach for general graphs is complete for the class NL of languages accepted by non-deterministic log-space machines, and remains NL-hard even if the

graphs are acyclic. For undirected graphs, it is complete for the class L of languages accepted by deterministic log-space machines (Reingold 2005), and is sandwiched between L and $UL \cap co-UL$ for planar directed graphs (Bourke et al. 2007). Here UL is the class of languages accepted by unambiguous non-deterministic log-space machines, and co-UL is the class of complements of languages in UL. Clearly, Distance and Long-Path are at least as hard as Reach: $(G, s, t) \in \text{Reach}$ if and only if $(G, s, t, n) \in \text{Distance}$ if and only if $(G, s, t, 0) \in \text{Long-Path}$, where n is the number of vertices in G . Distance is NL-complete for general graphs, and remains NL-hard even if the graphs are acyclic, or if the graphs are undirected (Tantau 2007), but it is in $UL \cap co-UL$ for planar directed graphs (Thierauf & Wagner 2008). Long-Path is NP-complete for general graphs, since it includes Hamiltonian paths as a special case. It remains NP-hard for planar undirected graphs. However, for directed acyclic graphs, there is a deterministic linear time algorithm based on dynamic programming. The problem is also known to be NL-complete for directed acyclic graphs. However its complexity for planar directed acyclic graphs is, to the best of our knowledge, not yet studied.

In this note we consider this combination of planarity and acyclicity for Long-Path. One of our main results is:

Theorem 1 PDLP, *the Long-Path problem for planar directed acyclic graphs, is in $UL \cap co-UL$.*

Thus Long-Path shares the current best-known upper bounds for Reach and Distance for planar DAGs. We also address the question of when the three problems are indeed equivalent on DAGs, and give partial bounds (Theorems 9,11). A recent result in (Jakoby & Tantau 2007) shows that for an important subclass of planar DAGs, namely series-parallel graphs, the three problems are indeed equivalent and are all L-complete. Theorem 1 is in fact an unobserved corollary of their construction (see also (Jakoby & Tantau 2006)). An analogous result for planar DAGs equating the three problems would be nice, but is not known.

We show that the search version of Long-Path shares the same upper bound for planar DAGs (Lemma16).

For graphs with embeddings on the torus, (Allender et al. 2005) shows that reachability is no harder than planar reachability. We observe that Distance and Long-Path are also no harder than the planar versions (Corollary 7).

We also address the problem of counting paths in directed acyclic graphs. The problem of counting the number of paths between two designated nodes s and t in a DAG is known to be complete for #L, the class of functions that count the number of accepting paths in NL machines. In (Allender et al. 1999)

Allender *et al.* considered a restriction of this problem when the number of such paths is bounded by a polynomial. They gave an NL upper bound for this problem. We consider this restriction combined with other restrictions and give orthogonal bounds. We consider the case when the DAG not only has polynomially bounded number of s to t paths but also t is the unique sink in the DAG. We show that in this case, the number of paths can be computed by a deterministic AuxPDA (a log-space machine augmented with a stack) running in polynomial time, DAuxPDA. It is known that such machines accept exactly the class LogDCFL of languages reducible via log-space many-one reductions to some deterministic context-free language (Sudborough 1978). Another restriction is planarity. We show that in planar DAGs where the number of s - t paths is bounded by a polynomial, this number can be determined by an unambiguous pushdown automaton that uses a log-space auxiliary work-tape and runs in polynomial time (UAuxPDA). For planar DAGs, we consider the problem of computing the number of longest or shortest s to t paths when this number is bounded by a polynomial, irrespective of the total number of paths. We show that this can be done by a UAuxPDA. Thus we have the following theorem, which we prove in Section 5.3:

Theorem 2 1. *The problems #PDLP_{poly} and #Path_{poly} for planar directed acyclic graphs can be solved by a UAuxPDA in polynomial time.*

2. *The #Path_{poly} problem for directed acyclic graphs is in LogDCFL when the paths to be counted are from some designated node s to the unique sink t .*

This paper is organised as follows. In Section 2 we state some known results that we use, and also describe preprocessing steps that we use in our algorithms. Section 3 describes the algorithm, based on (Jakoby & Tantau 2007) and (Thierauf & Wagner 2008), that establishes Theorem 1. Section 4 describes the alternative algorithm using double inductive counting. In Section 5, algorithms for the search version, and for the counting version under a polynomial bound promise, are described.

2 Preliminaries

For any subclass \mathcal{C} of graphs, let $\text{Reach}(\mathcal{C})$, $\text{Distance}(\mathcal{C})$, and $\text{Long-Path}(\mathcal{C})$ denote the restriction of these problems to instances from \mathcal{C} .

For directed acyclic graphs, $(G, s, t) \in \text{Reach} \Leftrightarrow (G, s, t, |V|) \in \text{Distance} \Leftrightarrow (G, s, t, 0) \in \text{Long-Path}$. So $\text{Distance}(\mathcal{C})$ and $\text{Long-Path}(\mathcal{C})$ are at least as hard as $\text{Reach}(\mathcal{C})$ for any subclass \mathcal{C} of directed acyclic graphs.

We use the following results:

Lemma 3 ((Bourke et al. 2007)) *Reach in planar directed graphs is in UL \cap co-UL.*

Lemma 4 ((Thierauf & Wagner 2008)) *Distance in planar directed graphs is in UL \cap co-UL.*

Lemma 5 ((Jakoby & Tantau 2007)) *Distance and Long-Path in series-parallel graphs are equivalent.*

Lemma 6 ((Allender et al. 2005)) *Reach(Torus) reduces via log-space many-one reductions to planar Reach.*

Consider any directed acyclic instance (G, s, t, k) of Distance or Long-Path. By (parallel) queries of the form (G, s, u) or (G, u, t) to Reach, we can remove all vertices that do not figure on some s -to- t path. Thus in $\mathbb{L}^{\text{Reach}}$ we can obtain a single-source (s) single-sink

(t) graph G' , and all queries to Reach involve only the graph G . So now onwards we only consider the case where we want to find a long path between the unique source and the unique sink.

If the input graph G is not planar but can be embedded on a torus, then we use the construction of Lemma 6. This gives a planar graph G' with the following properties: There are $l \in O(n)$ copies of G cut and stitched together, and hence there are l vertices t_1, \dots, t_l and one special vertex, say s_1 , such that

$$\exists \rho : s \rightarrow_G^* t \wedge |\rho| = l \Leftrightarrow \exists i \exists \rho_i : s_1 \rightarrow_{G'}^* t_i \wedge |\rho_i| = l$$

This gives a logspace truth-table reduction from toroidal to planar graphs, preserving path lengths. This can be converted to a many-one reduction using the technique from (Allender et al. 2005): there, formula-truth-table reductions to Reach(Planar) are converted to many-one reductions to Reach(Planar), but it is easy to see that the conversion preserves path lengths as well. Hence

Corollary 7

$$\text{Distance}(\text{Torus}) \leq_m^{\log} \text{Distance}(\text{Planar})$$

$$\text{Long-Path}(\text{Toroidal DAGs}) \leq_m^{\log} \text{Long-Path}(\text{Planar DAGs})$$

3 Algorithm using distance computation

Our first algorithm for Long-Path in planar DAGs uses a simple extension of Lemma 5.

Lemma 8 *Distance and Long-Path in planar directed acyclic graphs are equivalent modulo planar reachability.*

This actually follows from (Jakoby & Tantau 2007) itself; though they claim their result only for series-parallel graphs, it works for any class of acyclic graphs that is closed under subdivision and vertex deletion. In particular, it works for planar DAGs. To make this clear, we present below in Theorem 9 their proof simplified by specialising to unweighted graphs, and stated with minimum conditions. Lemma 8 is an obvious corollary.

Theorem 9 *Let \mathcal{C} be any subclass of directed acyclic graphs closed under subdivisions and vertex deletions. There is a function f , computable in L with oracle access to $\text{Reach}(\mathcal{C})$, that reduces $\text{Distance}(\mathcal{C})$ to $\text{Long-Path}(\mathcal{C})$ and $\text{Long-Path}(\mathcal{C})$ to $\text{Distance}(\mathcal{C})$.*

Proof: Let $G = (V, E)$ be the given directed acyclic graph, in which we want to find the longest, or the shortest, path between given vertices s and t . Recall from Section 2 that with oracle access to $\text{Reach}(\mathcal{C})$, we can assume that s is the unique source and t the unique sink. Every vertex of G lies on some $s \rightarrow^* t$ path. (G is unweighted, so all edges have weight 1.) Construct a new graph $G' = (V', E')$ as follows:

For each $u \in V$, define $P_u = \{x \in V \mid x \rightarrow_G^* u\}$. Since s is the unique source, $\forall u, s \in P_u$. Since t is a sink, $\forall u \neq t, t \notin P_u$. Also, $\forall u, u \in P_u$. Next define $E_u = \{(x, y) \mid x \in P_u, y \notin P_u\}$. Since G is acyclic, $\forall \langle x, y \rangle \in E, \langle x, y \rangle \in E_x$.

Let ρ be any $s \rightarrow^* t$ path. For every vertex $u \in V \setminus \{t\}$, $|\rho \cap E_u| = 1$. This can be seen as follows: $s \in P_u, t \notin P_u$, and along the path ρ , we transit from being in P_u to being outside P_u exactly once. Let this transition occur on edge $\langle x, y \rangle$. Then $\langle x, y \rangle \in E_u$, and no other edge of ρ can be in E_u .

To obtain G' , we replace each edge $e = \langle u, v \rangle$ by a path of length l_{uv} determined as follows:

$$\begin{aligned} l_{uv} &= 2 \left(\sum_{x \in V: \langle u, v \rangle \in E_x} \text{out-degree}(x) \right) - 1 \\ &= 2 \left(\sum_{x \in V: u \in P_x, v \notin P_x} \text{out-degree}(x) \right) - 1 \end{aligned}$$

Since G is acyclic, the vertex u itself always qualifies in the above sum, and so l_{uv} is positive.

Now the crucial claim: each $s \rightarrow^* t$ path ρ in G , of length $|\rho|$ in terms of number of edges, is transformed by the above to a path in G' of length *exactly* $2|E| - |\rho|$. This is because the length of the transformed path is

$$\begin{aligned} &\sum_{uv \in \rho} l_{uv} \\ &= \sum_{uv \in \rho} \left[2 \left(\sum_{x \in V: u \in P_x, v \notin P_x} \text{out-degree}(x) \right) - 1 \right] \\ &= 2 \left(\sum_{uv \in \rho} \sum_{x \in V: u \in P_x, v \notin P_x} \text{out-degree}(x) \right) - |\rho| \\ &= 2 \sum_{x \in V} \left(\text{out-degree}(x) \sum_{e \in \rho \cap E_x} 1 \right) - |\rho| \\ &= 2 \left(\sum_{x \in V} \text{out-degree}(x) \cdot |\rho \cap E_x| \right) - |\rho| \\ &= 2 \sum_{x \in V} \text{out-degree}(x) - |\rho| = 2|E| - |\rho| \end{aligned}$$

It thus follows that the longest (shortest) path in G is mapped to the shortest (longest, respectively) path in G' . In fact, if the $s \rightarrow^* t$ paths are ordered monotonically with respect to length, then the above transformation precisely reverses this ordering. Hence the reduction function f maps (G, s, t, k) to $(G', s, t, 2|E| - k)$.

The next crucial observation: G' can be obtained from G in logspace with oracle access to Reach , where all queries involve only the graph G . This is because obtaining G' merely involves finding the sets P_u, E_u . ■

Theorem 1 follows from Theorem 9 and Lemma 4.

4 Algorithm using inductive counting

There is another method to obtain Theorem 1, by-passing Theorem 9 but using Lemmas 3,4. We sketch it here because it is instructive to see how double inductive counting can be used, and also because it says something more general as well: it places Long-Path in $(\text{UL} \cap \text{co-UL}) \oplus \text{Reach}(C)$ for any family C of acyclic max-unique graphs (Theorem 11). By *max-unique* we mean that for each pair u, v , if there is a path from u to v , then the longest path from u to v is unique. (Similarly, a graph is *min-unique* if for each pair u, v , if there is a path from u to v , then the shortest path from u to v is unique.) Here \oplus represents languages expressible as the marked union of languages from the two classes. That is, for languages A and B , $A \oplus B$ is the language $\{0w \mid w \in A\} \cup \{1w \mid w \in B\}$, and for language classes \mathcal{C}_1 and \mathcal{C}_2 , their marked union is $\mathcal{C}_1 \oplus \mathcal{C}_2 = \{A \oplus B \mid A \in \mathcal{C}_1, B \in \mathcal{C}_2\}$.

The initial steps are similar to those used in (Thierauf & Wagner 2008) to place planar Distance in $\text{UL} \cap \text{co-UL}$.

1. Given a graph \hat{G} , make it single-source single-sink G as described in the preprocessing step. Reduce the degree of each vertex to 3. (To reduce the degree of nodes, in (Allender et al. 2005) a vertex of degree d is replaced by a cycle of length d . Since we cannot afford to introduce cycles, we use the trick of (Chakraborty & Datta 2006); insert incoming and outgoing trees at each vertex.) This construction maps edges to paths, and we can identify a unique new edge as "responsible" for each original edge. We mark such edges.

Embed G into a grid using the (Allender et al. 2005) reachability-preserving construction. The output of this step is a grid graph G' , with the edges of G (original edges) marked in G' and is obtained in logspace. If the original graph G had n vertices, the new grid graph is of dimensions $n^2 \times n^2$.

2. The graph G' is then subject to a weighting scheme building upon that of (Bourke et al. 2007), and can be described as follows: every horizontal edge e gets weight $n^4 + (\text{mark}(e) \times n^8)$, and every vertical edge e gets weight $n^4 + (\text{mark}(e) \times n^8) + (\text{up}(e) \times \text{col}(e))$, where $\text{mark}(e)$ is one if the edge e is marked; zero otherwise, $\text{col}(e)$ equals the column number in which the edge e appears, and $\text{up}(e)$ is $+1$ if the edge e is upwards, -1 otherwise. This is the graph G'' .

The weighting scheme ensures that G'' is min-unique.

3. The last step in (Thierauf & Wagner 2008) is to use the double counting technique of (Reinhardt & Allender 1997); this technique is applicable because G'' is min-unique. The idea here is to use the inductive counting counter c_k that keeps track of number of vertices within distance k , and to use a cumulative paths counter Σ_k that keeps track of the shortest paths of the nodes so counted. The first counter allows checking the complement of reachability, the second allows doing so unambiguously. As mentioned in (Thierauf & Wagner 2008), a third counter m_k tracking cumulative marked edges can be added, allowing distance computation uniquely.

The argument of (Thierauf & Wagner 2008) concerning Step 2 is restricted to shortest paths; however, one can observe something more general about the above weighting scheme.

Observation 10 *For any length l , all the st paths of length l in G will be mapped to paths of weight greater than $(l \times n^8)$ and less than $((l+1) \times n^8)$ in G'' , and the maximum weight and the minimum weight paths in this range will be unique. Thus G'' is both min-unique and max-unique.*

Observation 10 already guarantees a max-unique graph.

Step 3 above can not be used as it is. For computing the shortest path, in (Reinhardt & Allender 1997), the variables could be initialised as $c_0 = 1$ and $\Sigma_0 = 0$. If the same semantics is to be used for computing the longest path, then c_0 should be the number of vertices having length of the longest path from s at least 0, and should be initialised to n . However Σ_0 should then contain the total lengths of all the longest paths, which is an unknown quantity. To handle this, we redefine Σ_k to be sum of lengths of the longest

paths for those vertices whose longest path to t is of length at most k . This allows a procedure similar to (Reinhardt & Allender 1997) to work correctly, but now it is no longer unambiguous. To make it unambiguous, we introduce more nondeterminism into the (Reinhardt & Allender 1997) procedure. We guess the sum of lengths of all the $u \rightarrow^* t$ longest paths *a priori* and tally it in the end with the final Σ_k .

Thus we show

Theorem 11 *Let G be a directed acyclic graph with a unique source s , a unique sink t , such that G is max-unique. (For each pair u, v , if there is a path from u to v , then the longest uv path is unique.) Then the length of the longest st path can be computed in $UL \cap \text{co-UL}$.*

The detailed procedures are described in Algorithms 1, 4 and 4.

The proof follows from Claims 12, 13, 14 and 15 stated and proved below.

Notation: $D(v)$ = Length of the longest path from v to t .

$$S_k = \{v \mid D(v) \geq k\}, \quad c_k = |S_k|$$

$$\Sigma_k = \sum_{v \in V \setminus S_k} D(v), \quad T = \sum_{v \in V} D(v)$$

Algorithm 1 Main

Input: G, s, t
 Guess nondeterministically $M = \sum_{v \in V} D(v)$
 with $n \leq M \leq n^2$.
 $c_0 \leftarrow n, \Sigma_0 \leftarrow 0, k \leftarrow 0$
while $c_k \neq 0$ **do**
 $k \leftarrow k + 1$
 Update (c_k and Σ_k)
end while
if $\Sigma_k \neq M$ **then**
 Halt and reject
else
 Accept
end if

Algorithm 2 Update: Procedure for updating c_k and Σ_k

Input: $G, s, t, c_{k-1}, \Sigma_{k-1}$
 $c_k \leftarrow c_{k-1}, \Sigma_k \leftarrow \Sigma_{k-1}$
for all $v \in V$ **do**
 if $\text{Test}(G, k-1, c_{k-1}, \Sigma_{k-1}, v) = \text{true}$ **then**
 if for all out-neighbours x of v ,
 $\text{Test}(G, k-1, c_{k-1}, \Sigma_{k-1}, x) = \text{false}$ **then**
 $c_k \leftarrow c_k - 1, \Sigma_k \leftarrow \Sigma_k + k - 1$
 end if
 end if
 end if
end for

Claim 12 *If the guessed value of M is correct (i.e. $M = T$), then algorithm **Test**, given the correct values of c_k and Σ_k as input, reports a decision on exactly one path.*

Proof: The procedure **Test**, on each run R , guesses an $x \rightarrow^* t$ path R_x for each vertex x . Depending on its guess for $D(x) \geq k$, it adds the length of R_x to either sum or sum' . Finally these have to add up to M for **Test** to report a decision.

When $M = T$, M is indeed the sum of all $D(x)$. This can match $\text{sum} + \text{sum}'$ exactly when all the guessed paths R_x are longest. Since G is max-unique, this happens on exactly one run. ■

Algorithm 3 Test: An unambiguous procedure to test if $D(v) \geq k$

Input: G, k, c_k, Σ_k, v
 $\text{count} = n, \text{sum} = 0, \text{path.to.v} = \text{true}, \text{sum}' = 0$
for all $x \in V$ **do**
 Guess nondeterministically if $D(x) \geq k$
 if Guess is no **then**
 Guess a path of length $l < k$ from x to t .
 {If this fails then reject and halt.}
 $\text{count} \leftarrow \text{count} - 1$
 $\text{sum} \leftarrow \text{sum} + l$
 if $x = v$ **then**
 $\text{path.to.v} = \text{false}$
 end if
 else
 Guess a path of length $l' \geq k$ from x to t .
 {If this fails then reject and halt.}
 $\text{sum}' \leftarrow \text{sum}' + l'$
 end if
end for
if $\text{count} = c_k$ and $\text{sum} = \Sigma_k$
 and $\text{sum}' + \text{sum} = M$ **then**
 return path.to.v
else
 Reject and halt.
end if

Claim 13 *For any guessed value of M , given the correct values of c_k and Σ_k as input, all paths of algorithm **Test** that do not lead to rejection always return the correct decision.*

Proof: As described in the preceding proof, each run of **Test** guesses a path R_x for each x . It may guess a path of length shorter than $D(x)$, but not longer. Since count is decremented only when it guesses that $D(x) < k$, and for other guesses some witnessing path of length at least k is found, at the end the value of count is at most as large as c_k .

Suppose on some run **Test** returns a decision. Then on this run $\text{count} = c_k$. Suppose further that the decision is wrong.

Case 1: $D(v) < k$, but **Test** reports that it is larger. This cannot happen, since **Test** has to find a witnessing path of length at least k .

Case 2: $D(v) \geq k$, but **Test** reports that it is smaller. Then this run of **Test** does not account for v in count . So at the end of the run, $\text{count} < c_k$, a contradiction. ■

Claim 14 *If the queries ($D(v) \geq k$) are answered correctly by **Test**, then given c_{k-1} and Σ_{k-1} , the values of c_k and Σ_k are updated correctly by algorithm **Update**.*

Proof: **Update** starts by assuming that $S_k = S_{k-1}$ and so $c_k = c_{k-1}$. Note that $S_k \subseteq S_{k-1}$, so **Update** only has to detect when to remove vertices from its current S_k .

For each v , **Update** checks whether $D(v) \geq k - 1$ and $D(u) < k - 1$ for all out-neighbours u of v . If this holds, then the longest path from v to t is of length exactly $k - 1$ and $v \notin S_k$. Thus the procedure decrements c_k by 1 and increments Σ_k by $k - 1$.

So if all the queries are answered correctly by **Test**, then what **Update** does is correct. ■

Claim 15 *The algorithm **Main** is correct and unambiguous.*

Proof: **Main** starts with the correct values of c_0 and Σ_0 . From claims 13 and 14, the correctness of **Main**

is immediate. In particular, the final value of Σ_k is always correct.

If $M = T$, then by Claim 12, procedure **Test** always returns a decision, unambiguously. Thus exactly one path of **Main** (amongst those where $M = T$ was guessed) leads to a decision, and this decision is correct.

If $M > T$, then no run of **Test**, at any stage k , can trace paths adding up to M . So **Test**, and hence **Update**, and **Main** have no accepting run.

If $M < T$, consider the runs on which **Test** and **Update** proceed to finally compute Σ_k . Since **Main** is correct, we know that $\Sigma_k = T$. Now the check $M = \Sigma_k$ fails and **Main** rejects and halts. ■

Thus the method from (Thierauf & Wagner 2008), with step 3 replaced by the algorithms establishing Theorem 11, places acyclic planar Long-Path in $\perp \text{UL} \cap \text{co-UL}$ and hence $\text{UL} \cap \text{co-UL}$.

5 Extensions of the longest path algorithm

In this section we consider some variations of Long-Path: enumerating the path, finding the second longest path, counting the number of paths under some promise. We obtain bounds for planar DAGs and DAGs with other restrictions.

5.1 Enumerating the longest path

The algorithm in section 4 computes the length of the longest path in a planar DAG. Here we extend it to show that the search version also has the same bound. That is, we show

Lemma 16 *A longest path in a planar DAG can be found in $\perp \text{UL} \cap \text{co-UL}$ and hence in $\text{UL} \cap \text{co-UL}$.*

Recall that a planar DAG G is transformed to a max-unique planar DAG G'' via subdivisions and deletion of unreachable vertices. The unique longest path in G'' “corresponds” to some longest path in G . Hence, we retrieve the unique s - t path ρ'' in G'' and map it back to a longest path ρ in G .

If an edge $e = \langle u, v \rangle$ is on ρ'' , then removing e strictly reduces the length of the s - t longest path in the resulting graph $G''_e = G'' \setminus \{e\}$. Otherwise, G''_e still has ρ'' , so the longest s - t path length is the same in G''_e and in G'' . This allows us to design a log-space machine M_1 that on input G'' , outputs exactly the edges of ρ'' in some order. This machine first queries the $\text{UL} \cap \text{co-UL}$ oracle about the length l of the longest path in G'' , and stores l . Then it cycles through all edges e in G'' . It queries the oracle for the length l' of the longest s - t path in $G''_e = G'' \setminus \{e\}$. If l' is strictly less than l , it outputs e . This gives a listing of all the edges that lie on ρ'' .

A second log-space machine M_2 takes G and the output ρ'' of M_1 as input, and produces as output the corresponding path ρ in G .

Composing M_1 and M_2 gives the desired machine, proving Lemma 16.

5.2 Length of the second longest path

We consider another variation of Long-Path. Given a planar DAG G , the algorithm in Section 5.1 produces a longest path ρ . Can we find the (length of the) longest path other than ρ ? Note that this may well have the same length as ρ , because G itself need not be max-unique.

The algorithm described in section 5.1 converts G to a max-unique graph G'' . To find the longest path ρ'' of length l in G'' , it makes queries to Long-Path in

the graphs $G'' \setminus \{e\}$ for each edge e . Each query returns a length l' . The maximum among the l' values obtained is the length of the longest path other than ρ'' in G'' ; call this path σ'' . Then the corresponding path σ in G is in fact the longest path in G other than ρ . Thus, to find $|\sigma|$, the machine just needs to maintain the maximum over all the l' values obtained through queries, thus obtaining $|\sigma''|$, and map this back to $|\sigma|$; all of this can be done in log-space. To find σ itself, the machine needs to maintain the pair $\langle l', e \rangle$ where l' is the maximum value and is obtained in the graph $G'' \setminus \{e\}$. Then re-using the search algorithm on $G'' \setminus \{e\}$ gives σ'' and hence σ .

This can be generalised to finding the (length of the) k^{th} longest path, as long as k is a constant. Thus we have:

Lemma 17 *For constant k , given a planar DAG G and vertices s, t in it, k longest paths from s to t in G can be found in $\text{UL} \cap \text{co-UL}$.*

Remark: The k longest paths may not be unique due to ties. The algorithm guarantees that any $s - t$ path not listed by it is no longer than the shortest $s - t$ path listed by it.

5.3 Computing the number of polynomially bounded paths

The problem of counting the number of paths between two designated nodes in a DAG is known to be complete for $\#\text{L}$. Consider a restriction of this problem when the number of such paths is bounded by a polynomial. With this restriction it is natural to believe that the counting problem is easier, because even **Reach**, which is otherwise NL -complete, is in LogDCFL for such graphs (Buntrock et al. 1991). In (Allender et al. 1999) Allender *et al.* substantiated this belief by giving an NL upper bound for the counting problem under this restriction. We consider a further restriction where the DAG not only has polynomially bounded number of s to t paths but also t is the unique sink in the graph, and show the following:

Theorem 18 *Given a DAG G and vertices s, t where t is the unique sink in G and the number of s - t paths is bounded by a polynomial in the size of G , the number of s - t paths can be computed by a DAuxPDA in polynomial time, and hence is in LogDCFL .*

See Algorithm 4.

The algorithm explores the DAG starting from s . The number of s - t paths explored is stored in a variable *count*. It assumes an ordering on the labels of the vertices (lexicographical ordering will suffice) and an additional label 0 which is assumed to be the smallest in the ordering. It traverses the graph in a depth-first manner, putting visited vertices on the stack. The label of the vertex being visited in the current step is stored in the variable *current*. The traversal is started from s , taking the out-neighbour with the smallest label at each step. Whenever t is reached, *count* is incremented.

On reaching t , the algorithm backtracks by popping the stack, retrieving the vertex v visited just before t . The label of t is stored in a variable *previous* and the label of v is stored in *current*. If v has an out-neighbour u with a label larger than *previous*, the traversal is continued along the (v, u) edge, and the label of this out-neighbour is stored in the variable *next*. Otherwise, the backtracking process is continued by popping the stack again, storing the label of v in *previous*, and setting *current* to the newly popped label.

Thus, at any point of time, the stack contains nodes on the path from s to $current$ (excluding $current$), and the vertex $current$ is being explored in the forward or backward direction.

Algorithm 4 Procedure to count s to t paths in a single sink DAG, where t is the sink.

Input: DAG G with designated nodes s and t , 1^k
Question: Is the number of s to t paths exactly k ?

```

1:  $current \leftarrow s, previous \leftarrow 0, count \leftarrow 0.$ 
2: repeat
3:    $next \leftarrow$  smallest out-neighbour of  $current$ 
     larger than  $previous.$ 
4:   if  $next = t$  then
5:      $count \leftarrow count + 1.$ 
6:     if  $count > k$  then
7:       reject and halt.
8:     end if
9:      $previous \leftarrow t.$ 
10:  else if  $next = NULL$  then
11:     $previous \leftarrow current.$ 
12:     $current \leftarrow pop(top-of-stack).$ 
13:  else
14:     $push(current).$ 
15:     $current \leftarrow next.$ 
16:     $previous \leftarrow 0.$ 
17:  end if
18: until  $current = s$  and  $next = NULL$ 
19: if  $count = k$  then
20:   Accept.
21: else
22:   Reject.
23: end if

```

Lemma 19 *Algorithm 4 correctly computes the number of s to t paths.*

Proof: We prove a stronger claim here: Algorithm 4 explores all s - t paths in lexicographically increasing order, and each path is explored exactly once.

We represent an s - t path ρ as a sequence of labels of nodes (s, u_1, \dots, u_r, t) in the order in which they appear in ρ . As the graph is a DAG, each node appears at most once on ρ . We prove the claim by induction on the number of paths that have already been explored at some point of time during the execution of the algorithm. As the base case, the first path is chosen by picking the smallest out-neighbour of s , say v_1 , then the smallest out-neighbour of v_1 and so on. Thus the first path is the lexicographically smallest s - t path.

Now assume that the algorithm has explored j lexicographically smallest s - t paths, in lexicographically increasing order, the last one being $\rho = (s, u_1, \dots, u_r, t)$. When t is reached along ρ , the stack contains labels of nodes s, u_1, \dots, u_{r-1} . The variables $current$ and $next$ are s and t respectively. The algorithm checks if u_r has an out-neighbour u_{r+1} with label larger than that of t . Otherwise it pops the stack till a node u_i is found that has an out-neighbour u'_{i+1} with label larger than that of u_{i+1} . It picks the smallest such u'_{i+1} and starts exploring a new path ρ' . Thus ρ' shares a prefix s, u_1, \dots, u_i with ρ . As t is the only sink and the graph is a DAG, each such exploration ends in t .

We claim that ρ' is the s - t path which is lexicographically next to ρ . Assume that there is another path ρ'' such that $\rho < \rho'' < \rho'$ lexicographically. Then ρ'' should share a longer prefix $s, u_1, \dots, u_i, \dots, u_k$ with ρ , i.e. $\rho'' = (s, u_1, \dots, u_k, u'_{k+1}, \dots, t)$ such that $label(u_{k+1}) < label(u'_{k+1})$. But then the algorithm

would have encountered u_k before u_i while backtracking after finding ρ . This contradicts the assumption that the algorithm picks the last node on ρ that has an out-neighbour with label larger than the label of its out-neighbour on ρ . Thus ρ' is the lexicographically next path after ρ .

Thus the algorithm always explores the path that is lexicographically next to the most recently explored path, and hence no path is explored twice. When the algorithm explores the lexicographically largest path, it can not find any such u_i from which a new path can be constructed. In this case, all the nodes on the stack are popped and the algorithm stops. ■

Lemma 20 *In Algorithm 4, for each s to t path ρ , each edge on ρ is traversed exactly once in the forward direction. During the backtracking phase, after reaching t along ρ , an edge on ρ is traversed at most once in the backward direction. Consequently, Algorithm 4 can be implemented in LogDCFL.*

Proof: From Lemma 19, it is clear that each path is explored exactly once. Thus, for each s - t path ρ , each edge on ρ is traversed exactly once in the forward direction. During backtracking, only the nodes on the stack are popped one by one, and no new edge is traversed. Thus an edge can be traversed at most once during backtracking. During backtracking, if a node u is encountered (before popping s) which has a larger out-neighbour than its out-neighbour on ρ , then the backtracking stops, and exploration of a new path is started. In this case, edges that form a prefix of ρ (from s to u) still remain on stack, and they are not traversed backwards for ρ .

As the number of s - t paths is polynomially bounded, and each path being explored ends in t , the algorithm runs in polynomial time.

Apart from the stack, the algorithm needs to store only three labels $current, previous, next$ at any point of time. These can be stored in log space. It also needs to store the value of $count$. This can also be stored in log space since the value of $count$ is always bounded by a polynomial.

Thus the algorithm can be implemented by a polynomial time DAuxPDA, and hence in LogDCFL. ■

Theorem 18 follows from Lemmas 19, 20.

5.4 Counting polynomially bounded paths in planar DAGs

We now consider applying Theorem 18 to planar DAGs.

Algorithm 4 may not be directly applicable, since the given DAG may have multiple sinks. These sinks can be removed in $UL \cap co-UL$ by queries to *Reach*, to get a planar DAG with t as the only sink. Then Algorithm 4 can be used. Overall, the combined algorithm uses logspace in a nondeterministic unambiguous manner, and a stack in a deterministic manner, giving an algorithm that can be implemented on a UAuxPDA in polynomial time. Thus

Corollary 21 *Given a planar DAG G and vertices s, t where the number of s - t paths is bounded by a polynomial in the size of G , the number of s - t paths can be computed by a UAuxPDA.*

Our algorithm is similar to that given by (Buntrock et al. 1991) for testing reachability in the computation tree of an NL machine when each configuration of the machine is reachable by polynomially many paths.

Now consider the problem of computing the number of longest s - t paths when this number is bounded by a polynomial. Note that the total number of s to

t paths may not be polynomially bounded, hence just removing the sinks other than t as above does not suffice.

Algorithm 5 describes a procedure that removes all the edges which do not appear on any longest path from s to t . It finds out the length of the longest s - t path in the planar DAG G in $\text{UL} \cap \text{co-UL}$ using Theorem 1. Then G is layered and edges that are not a part of any s - t longest paths are removed. These are exactly the edges that go across more than one layer. Thus, in the resulting DAG, all s to t paths are the longest s to t paths. Moreover, t is the only sink in this new DAG.

Now we can use Algorithm 4 for counting s to t paths in this graph.

Algorithm 5 Procedure to delete all s to t paths shorter than the longest path.

Input: Planar DAG $G = (V, E)$, two designated vertices s, t ,
Output: Graph G' that has no paths other than s to t longest paths.
for all $v \in V$ **do**
 $\text{layer}(v) \leftarrow$ length of s to v longest path.
end for
for all edges (u, v) **do**
 if $\text{layer}(v) - \text{layer}(u) = 1$ **then**
 Output (u, v) .
 end if
end for

If instead of longest paths, we wish to count shortest paths, and this number is bounded by a polynomial, a similar approach can be used. The distance of a node v from s should be used as its layer number in Algorithm 5 instead of the length of the longest path from s to v . From Lemma 4 we know that distance can be computed in $\text{UL} \cap \text{co-UL}$.

Combining the $\text{UL} \cap \text{co-UL}$ layering procedure of Algorithm 5 with the DAuxPDA procedure of Algorithm 4, we have:

Corollary 22 *Given a planar DAG G and vertices s, t such that the number of distinct longest (or shortest) paths from s to t is bounded by a polynomial, this number can be determined by a UAuxPDA in polynomial time.*

As mentioned in Section 1, this number can also be determined in NL, and Corollary 22 gives an orthogonal bound.

References

- Allender, E., Datta, S. & Roy, S. (2005), The directed planar reachability problem, in ‘Proc. 25th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)’, LNCS, pp. 238–249.
- Allender, E., Reinhardt, K. & Zhou, S. (1999), ‘Isolation, matching and counting uniform and nonuniform upper bounds’, *Journal of Computer and System Sciences* **59**(2), 164–181.
- Bourke, C., Tewari, R. & Vinodchandran, N. V. (2007), Directed planar reachability is in unambiguous logspace, in ‘Proc. IEEE Conference on Computational Complexity (CCC)’, pp. 217–221.
- Buntrock, G., Jenner, B., Lange, K.-J. & Rossmanith, P. (1991), Unambiguity and fewness for logarithmic space, in ‘Proc. 8th International Symposium on Fundamentals of Computation Theory (FCT)’, Vol. 529 of LNCS, pp. 168–179.

Chakraborty, T. & Datta, S. (2006), One-input-face MPCVP is hard for L, but in LogDCFL, in ‘Proc. 26th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)’, Vol. 4337 of LNCS, pp. 57–68.

Jakoby, A. & Tantau, T. (2006), Computing shortest paths in series-parallel graphs in logarithmic space, in ‘Complexity of Boolean Functions’, Dagstuhl Seminar Proceedings. <http://drops.dagstuhl.de/opus/volltexte/2006/618>.

Jakoby, A. & Tantau, T. (2007), Logspace algorithms for computing shortest and longest paths in series-parallel graphs., in ‘Proc. 27th annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)’, Vol. 4855 of LNCS, pp. 216–227. see also (Jakoby & Tantau 2006).

Reingold, O. (2005), Undirected st -connectivity in logspace., in ‘Proc. 37th ACM Symposium on Theory of Computing (STOC)’, pp. 376–385.

Reinhardt, K. & Allender, E. (1997), Making nondeterminism unambiguous, in ‘IEEE Symposium on Foundations of Computer Science (FOCS)’, pp. 244–253.

Sudborough, I. (1978), ‘On the tape complexity of deterministic context-free language’, *Journal of Association of Computing Machinery* **25**(3), 405–414.

Tantau, T. (2007), ‘Logspace optimization problems and their approximability properties’, *Theory of Computing Systems* **41**(2), 327–350.

Thierauf, T. & Wagner, F. (2008), The isomorphism problem for planar 3-connected graphs is in unambiguous logspace, in ‘Proc. Symposium on Theoretical Aspects of Computer Science (STACS)’, Vol. 08001 of *Dagstuhl Seminar Proceedings*, pp. 633–644.