Space Complexity of Reachability in Simple Path Graphs

Krishnamoorthy Dinesh
Department of Computer Science and Engineering, Indian Institute of Technology Palakkad, India

Chandana Sasidharan
Department of Computer Science and Engineering, Indian Institute of Technology Palakkad, India

Abstract

One of the central questions in space complexity is whether nondeterministic logspace computations (NL) can be simulated in unambiguous logspace (UL). A stronger notion, *reach unambiguity* (ReachUL $\subseteq$ UL $\cap$ coUL), requires every configuration reachable from the start to have exactly one computation path [7]. It is known that directed graph reachability is NL-complete, planar graph reachability is in UL, and undirected graph reachability is in deterministic logspace (L). In this work, we study the space complexity of reachability problem for restricted directed graph families and show the following.

1. For reach unambiguous graphs (graphs with at most one path from start vertex to every vertex), Lange [13] showed that reachability is in ReachUL. As our main result, we show that for simple path graphs (introduced in [11], which contains reach unambiguous graphs), where each vertex has at most one *simple* path from the start, the reachability problem lies in UL $\cap$ coUL. The key difficulty lies in recognizing whether the input graph is a simple path graph or not.
2. Our first result can also be equivalently stated as follows: the recognition problem for simple path graphs is in UL $\cap$ coUL if and only if the reachability problem restricted to simple path graphs is also in UL $\cap$ coUL. Inspired by this, we investigate the complexity of graph recognition versus graph reachability for other directed graph classes. Observe that for any graph class, solving reachability (for the class) also solves the recognition problem for that class. We show that for reach unambiguous graphs, solving recognition is as hard as solving reachability (making both of them ReachUL-complete).

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

Keywords and phrases Space complexity, Graph reachability, Simple path graphs, Reach unambiguity, Unambiguity, UL

Digital Object Identifier 10.4230/LIPIcs.MFCS.2026.87

Funding *Chandana Sasidharan*: CS thanks the Department of Computer Science and Engineering, IIT Palakkad for the extended financial support received.

Acknowledgements We are grateful to the anonymous reviewers for their comments and suggestions and, in particular, for pointing out to us that recognizing simple path graphs also solves the reachability problem unambiguously (see Page 3).

1 Introduction

A central question in space complexity asks if nondeterministic computation (captured by the class NL consisting of languages accepted by machines using logspace with potentially multiple accepting paths on any run) can be made unambiguous (captured by the class UL consisting of languages accepted by machine using logspace with at most one accepting path on any run) [15]. A stricter form of unambiguity, namely, *reach unambiguity* (captured by the class ReachUL $\subseteq$ UL $\cap$ coUL) introduced in [7] requires that for every configuration reachable (which can be accepting, rejecting or aborting) from the start configuration, there

is exactly one path in the configuration graph of the machine[1]. Note that there are two existing definitions for ReachUL. Buntrock *et al.* [7] says that the configuration graph of the run of a reach unambiguous machine could have multiple accepting configurations, while Lange's [13] definition has a fixed unique accepting configuration. Both definitions yield the same class [7]. In this work, we use Lange's definition.

Figure 1 gives a visual representation of how the configuration graph of machines associated with each of the above complexity classes looks like.

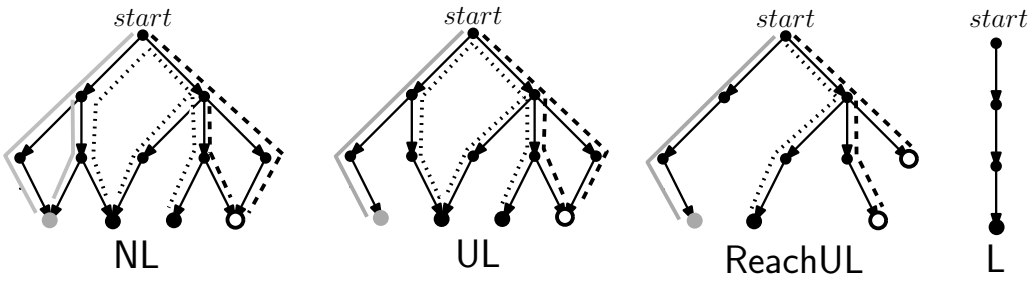

Figure 1 The gray vertices denote the accepting configuration, solid black vertices denote reject configuration, and black outlined vertices denote abort configuration. The gray paths are accepting paths, dotted paths are rejecting, and dashed paths are aborting. The path constraints are as follows: NL allows an arbitrary number of paths, UL requires at most one accepting path, ReachUL requires at most one accept, reject, and abort paths while L has exactly one computational path ending in either accept or reject configuration.

A natural way to get a better understanding of these classes is by studying problems complete for them (under many-one logspace reductions). While the general problem of directed graph reachability is complete for NL, we do not know of any natural complete problem for UL. Surprisingly, for the stricter notion of reach unambiguity, Lange [13] showed that reachability for a certain restricted family of graphs is complete for ReachUL (under logspace reductions). Of particular interest in this work is a special family of directed graphs called reach unambiguous graphs (where there is at most one path from the start vertex to every other vertex) for which the reachability problem is in ReachUL. This family of graphs is of interest as it is the only family of directed graphs for which reachability for n vertex graphs can be checked deterministically in $O(\frac{\log^2 n}{\log\log n})$ space, beating what is achievable with Savitch's theorem (For more details, see [2, 1]. For connections to logspace algorithms in the quantum setting, see [4]). Also, showing that a problem is in ReachUL implies that it can be solved in simultaneous polynomial time and poly-logarithmic space, which places it in the class SC^2 [7, 8]. It is an open problem to design such algorithms solving reachability for general directed graphs [16].

In this context, Kannan *et al.* [11] considered a strict generalization of this family called *simple path* graphs, where there is at most one *simple* path from the start vertex to any other reachable vertex[2]. They gave a simultaneous sub-linear space and polynomial time algorithm for solving graph reachability restricted to this class. Towards improving this result, if we could show that the reachability for simple path graphs is solvable reach unambiguously in logspace, then it gives a polynomial time and poly-logarithmic space algorithm (as $\mathsf{ReachUL} \subseteq \mathsf{SC}^2$).

[1] See Section 2 for a formal definition

[2] This is indeed a strictly larger family, as the vertices reachable from the start vertex must form a tree in reach unambiguous graphs while a simple path graph is allowed to contain directed cycles.

Towards this, as our first contribution, we show that the reachability problem for simple path graphs (denoted SimplePath-Reach) is in UL ∩ coUL (Theorem 1). Note that all the graphs considered in this work are directed.

▶ **Theorem 1** (Main). *Given a graph, checking if it is a simple path graph, is in* UL∩coUL*. In particular, this means that the reachability problem restricted to this class* SimplePath-Reach ∈ UL ∩ coUL*.*

Observe that if a graph G with s as a start vertex is guaranteed to be a simple path graph, then there is at most one simple path from s to any other vertex. Hence, such a path must also be the shortest path, which is unique. A result of Reinhardt and Allender [15] shows that given a graph G and two vertices s and t, if there is a logspace computable weight assignment to the edges of a graph that guarantees that the minimum weight path from s to t is unique, checking reachability from s to t in G is in UL ∩ coUL. Hence, assigning unit weights to each edge (which is indeed logspace computable) immediately shows that the reachability problem SimplePath-Reach is in UL ∩ coUL. Hence, the difficult part is in recognizing whether the given graph is indeed a simple path graph[3]. In such graphs, the vertices that are *not* reachable from the start vertex could have arbitrary structure (see Figure 2). This also adds to the difficulty of the recognition problem for simple path graphs.

Due to this, it is not clear if one could design a (logspace computable) weighting scheme to check if a given graph is a simple path graph (via an application of Reinhardt and Allender's result). Instead, we build on the algorithm of Lange [13] for reach unambiguous graphs (which can be viewed as a depth bounded depth first search) and extend the same to simple path graphs.

Motivated by the relation between the recognition and reachability for simple path graphs, as our second contribution, we explore the relation for other graph classes also. In general, the complexity of reachability restricted to a specific family of graphs seems to give a lot of information on the power and limitations of space bounded complexity classes. A celebrated result due to Reingold [14] showed that if the graphs are undirected, then the reachability problem is solvable in logspace. On the other hand, the problem stays complete for NL if the graphs are restricted to be acyclic. Interestingly, the reachability problem is solvable in UL if the graph family is planar [6] (or even of higher genus [10]).

In this context, it is natural to ask about the complexity of the reachability problem for a special family of graphs $\mathcal{G}$, called the $\mathcal{G}$-Reach problem. For such a problem, it is also necessary to check if the graph belongs to the family of graphs $\mathcal{G}$, called as the $\mathcal{G}$-Rec problem. Note that, in general, the reachability problem for a family of graphs can be harder than the recognition problem. For instance, if $\mathcal{G}$ is the family of directed bipartite graphs $G = (S, T, E)$, where the edges may go from S to T, T to S or both, checking if the underlying undirected graph G is bipartite is in logspace [3], while reachability restricted to bipartite graphs is complete for NL. We ask if the same holds true for complexity classes below NL.

When $\mathcal{G}$ is reach unambiguous graphs, we show that the recognition problem is as hard as the reachability problem (Proposition 2), thereby exhibiting a class below NL for which the previous statement is not true. Note that, since $\mathcal{G}$-Reach for reach unambiguous graphs is in ReachUL [13], and as the recognition problem reduces to it, the recognition problem for reach unambiguous graphs is also solvable in ReachUL. Hence the recognition and reachability problems reduces to each other for reach unambiguous graphs.

[3] We thank the anonymous reviewer for pointing this out to us.

▶ **Proposition 2.** *Let $\mathcal{G}$ be reach unambiguous graphs, let M be a reach unambiguous machine by which $\mathcal{G}$-*Rec *is in* ReachUL*. Then, there exists a reach unambiguous machine which makes exactly one oracle query to $L(M)$, solving the $\mathcal{G}$-*Reach *problem.*

Towards proving the above, note that a ReachUL algorithm with a ReachUL oracle access can be simulated in ReachUL. Such a result is known when the base machine is in deterministic logspace [7]. We state this as a proposition.

▶ **Proposition 3.** $\mathsf{ReachUL}^{\mathsf{ReachUL}} = \mathsf{ReachUL}$

Indeed, $\mathsf{ReachUL} \subseteq \mathsf{ReachUL}^{\mathsf{ReachUL}}$. To see the other direction, any machine N with an oracle access to L' in ReachUL, where N accepts a language in ReachUL can be simulated reach unambiguously. As $\mathsf{ReachUL} = \mathsf{coReachUL}$ [7], the answers to oracle queries in L' can be found reach unambiguously.

Our techniques and an overview of proofs

We describe the main ideas that goes in proving Theorem 1. We start by describing Lange's procedure and highlight the challenges involved in solving reachability for simple path graphs.

Lange's procedure can be viewed as a depth bounded depth first on an input graph G. More precisely, the procedure proceeds in rounds (starting with $d = 1$), where each round starts by performing a pre-order traversal of the subgraph of G consisting of all the vertices that are reachable from the start vertex up to a fixed depth d. To facilitate the pre-order traversal, the procedure takes advantage of the graph structure and uses nondeterminism to correctly identify the next vertex in the pre-order traversal. If the nondeterministic choices are correct, the accepts and rejects will be unambiguous. The non-trivial part of Lange's algorithm is that if the nondeterministic choices are incorrect, the procedure aborts and does so unambiguously, making the overall algorithm reach unambiguous.

For each value of d, the procedure verifies that the above subgraph (vertices reachable from s at distance at most d) is indeed reach unambiguous. The run of this procedure eventually defines a DFS tree rooted at s consisting of all the vertices reachable from s. Note that this tree is never stored explicitly. We now discuss the challenges involved in adapting Lange's procedure to simple path graphs. Simple path graphs are those graphs for which any DFS tree starting from s does not have a forward or cross edges (See Lemma 10). It can be argued that Lange's procedure [13] detects if the input graph has a forward, back edge or cross edge and rejects in all the three cases unambiguously. In particular, this means that the procedure will reject simple path graphs.

The first challenge is to make sure that the procedure does not reject on a back edge. This requires us to unambiguously identify if an edge is a back edge or not. We achieve this using the procedures IsLeftChildBackEdge (Algorithm 1) and IsRightChildBackEdge, which takes in a vertex y and height h of y in the DFS tree and checks if its left or the right child forms a back edge unambiguously. However, the procedure need not abort unambiguously. Note that Lange's algorithm [13] does not need to perform such checks.

The second challenge is to identify the next vertex to visit in the traversal. Consider the case where the subgraph reachable from s is a tree with back edges. Suppose that the current vertex (being explored) has a left child which is not part of a back edge. Then, the pre-order traversal must next visit the left child. In a different case, if the current vertex is the left child of the parent and is also a leaf vertex, then the next vertex to visit in the

pre-order is the right child of the parent. However, it could be that the right child is a back edge[4]. Lange's procedure does not have to handle this case as ReachUL configuration graphs do not have back edges.

To understand the third challenge, let us describe how the next vertex in the pre-order traversal is found if there are no more left or right child remaining to move. This is because the left and right children have already been traversed or because they are back edges or lead to a deeper depth. Consider the case when the current vertex is a leaf and is the right child of its parent. Additionally, let the parent be the left child of its grandparent. In this case, the next vertex to visit is the grandparent's right child. However, it could be that this child is part of a back edge. We handle both of these issues in the Next and PathUncle procedure (Algorithm 3, Algorithm 4). The rest of this paper is organized as follows.

Organization of the paper

After giving the necessary background in Section 2, we first show that reachability in simple path graphs is in $\mathsf{UL} \cap \mathsf{coUL}$ in Section 3, proving Theorem 1 from Introduction. Towards this, we introduce a few procedures such as the Main (explained in Section 3.2) that performs a depth bounded depth first search on the input graph and checks for reachability. For the traversal, Main uses the procedure Next (explained in Section 3.3) that returns the pre-order successor of a given vertex in a special subgraph induced on the input graph. Two supporting subroutines used in Next are explained in Sections 3.1 and 3.4. Finally, we conclude with a discussion on the recognition vs reachability for special family of graphs in Section 4, proving Proposition 2 from Introduction.

2 Preliminaries

In this section, we give an overview on the complexity classes and graph theory terms used. The standard definitions for complexity classes appear in Kozen [12] and Arora-Barak [5]. As mentioned in the introduction, all the graphs considered are directed. One such graph class discussed in this work is reach unambiguous graphs. It is defined as follows in [13].

▶ **Definition 4** (Reach Unambiguous Graph (G, s))**.** *A directed graph G is a reach unambiguous graph if there exists at most one path from the start vertex to any other vertex in G. Moreover, the out-degree of each vertex is either 0 or 2 and each vertex is marked as internal or leaf.*

A strict generalization of Reach unambiguous graphs are Simple Path Graphs introduced in [11] (Definition 5). We also need the definition of $G_d(s)$ (Definition 6) and uncle of a vertex in $G_d(s)$ (Definition 7).

▶ **Definition 5** (Simple Path Graph)**.** *For a directed graph G and a vertex s, the pair (G, s) is a simple path graph if there exists at most one simple path from s to any vertex in the graph. Moreover, the out-degree of each vertex is either 0 or 2 and each vertex is marked as internal or leaf. For a vertex v with non-zero out-degree, $R(v)$ is its right child and $L(v)$ is its left child of vertex.*

Note that the graph induced by the unreachable vertices from s does not have any restrictions on its structure. Moreover, unlike reach unambiguous graph, simple path graphs may have walks from s to any other vertex reachable in G. Additionally, when it is clear from the context, vertex s is omitted while describing a simple path graph.

[4] Forward and cross edges are not possible by Lemma 10.

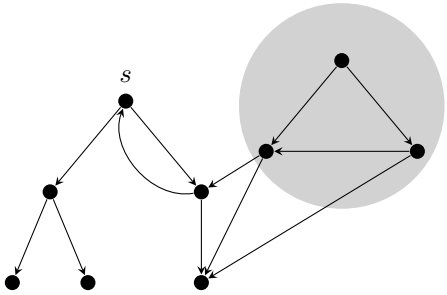

Figure 2 A simple path graph (G, s). The shaded region in gray denotes the unreachable vertices from s.

▶ **Definition 6** ($G_d(s)$ of a simple path graph G)**.** *For a simple path graph (G, s), $G_d(s)$ is the induced subgraph of G formed by all vertices reachable from s within distance d.*

▶ **Definition 7** (Uncle of a vertex v for a depth d)**.** *For a simple path graph (G, s), an uncle of a vertex v, $Uncle(v)$, is defined as follows. If v is s, uncle is $R(v)$. Otherwise, uncle is the next vertex to be visited in the pre-order traversal of subtree $G_d(s)$, assuming all the vertices of the subtree of $G_d(s)$ rooted at v are visited. See Figure 3 for an example.*

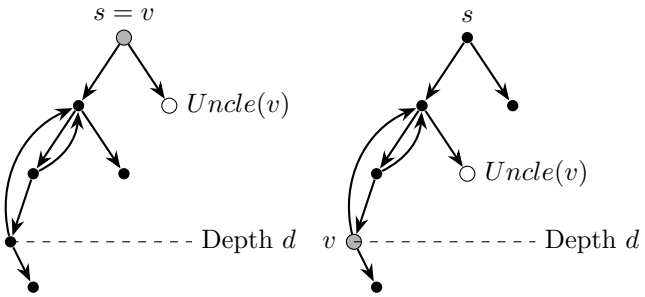

Figure 3 Example illustrating the uncle of a vertex v at different positions.

▶ **Definition 8** (DFS Tree, Tree edges)**.** *A DFS tree G' of a graph G for a DFS traversal (that is, pre-order traversal) is a subgraph of G with $V(G') = V(G)$ and $E(G')$ being the set of edges taken during the traversal. These edges are also known as tree edges.*

In a DFS tree of a directed graph G, the edges of G graph are classified as follows. This well known [9] formalization is described below.

▶ **Definition 9** (Edge classification in DFS)**.** *In a depth-first search (DFS) tree of a directed graph, edges in G not appearing in the DFS tree can be classified as follows. A forward edge is an edge that connects a vertex to a descendant in the DFS tree, but not directly to its immediate child. A cross edge is an edge that connects two vertices where neither is a descendant of the other. Finally, a back edge is an edge that points from a vertex to one of its ancestors in the DFS tree. This edge indicates the presence of a directed cycle in a directed graph. The dashed edge in Figure 2 is a back edge.*

Note that trees, by definition, do not contain forward, cross or back edges. Simple path graphs can be viewed as generalizations since they cannot contain forward or cross edges (but may contain back edges). We use this characterization crucially in Section 3.

▶ **Lemma 10** (Lemma 2 of [11]). *A pair (G, s) where G is a directed graph and $s \in V(G)$ is a simple path graph iff for any DFS tree produced from any vertex v reachable from s does not contain forward or cross edges.*

Complexity Preliminaries and Definitions

We begin by recalling two unambiguous logspace complexity classes that play a central role in our results. The class UL captures nondeterministic logspace computations with a unique accepting computation path, while ReachUL imposes the stronger requirement of unambiguity, allowing at most one computation path from the start configuration to any configuration. These notions are formalized below.

▶ **Definition 11** (UL). UL *is the complexity class containing decision problems that can be solved by a nondeterministic TM using a logarithmic amount of memory space and on every input have at most one computation path from the start configuration to any accepting configuration.*

▶ **Definition 12** (ReachUL [7]). ReachUL *is the complexity class containing decision problems that can be solved by a nondeterministic TM using a logarithmic amount of memory space and on every input have at most one computation path from the start configuration to any other configuration.*

Now, we consider the language L_{ru} defined in [13] where it was shown that it is ReachUL-complete.

▶ **Definition 13** (L_{ru}). *The language L_{ru} describes the family of graphs G together with a vertex labeling of G and a source vertex s with the property that from the source node s to every other vertex in G, there is at most one path. Moreover, the vertex labeling marks each node as either a leaf or an inner node, with children specified as left or right.*

For a structured family of graphs $\mathcal{G}$, we consider the problem of checking whether a graph belongs to a graph family $\mathcal{G}$ given as $\mathcal{G}$-Rec $= \{G \mid G$ is a graph and $G \in \mathcal{G}\}$) as well as the graph reachability problem restricted to this family of graphs $\mathcal{G}$-Reach $= \{(G, s, t) \mid G \in \mathcal{G}$-Rec and there is an s to t path in $G\}$.

3 Reachability in Simple Path Graphs is in UL intersection coUL

In this section, we show that reachability in simple path graphs is in $\mathsf{UL} \cap \mathsf{coUL}$ (Theorem 1). Before going into the details of the algorithm, we give a brief background for the problem.

Note that the simple path graphs introduced by [11] is a strict generalization of the graphs considered in L_{ru} (see Definition 13), which is the set of all graphs such that there is at most one path from start to any vertex in the graph. [11] showed that for any $0 < \epsilon < 1$, recognition as well as the reachability problem in simple path graphs can be done in $\mathsf{poly}(n)$ time and $\tilde{O}(n^{\epsilon})$ space where the $\tilde{O}$ hides the polynomial in $\log(n)$ factors. In this section, we build on an algorithm due to Lange [13]. Lange's algorithm checks whether G belongs to L_{ru}, reach unambiguously, and we extend it to our setting. In the next section, we start by discussing how the back edges are detected unambiguously.

3.1 Detecting Back Edges Unambiguously (Algorithm 1)

In this section, we first describe the procedure IsLeftChildBackEdge that takes in a vertex y and its height h in the subgraph rooted at s and checks whether $(y, L(y))$ or $(y, R(y))$ is a back edge, respectively. Let $G_h(s)$ denote this subgraph. The procedure assumes $G_h(s)$ is a simple

path graph and the correct height of y is h. These checks are not a part of the procedure. The algorithm for IsRightChildBackEdge is exactly same as that of IsLeftChildBackEdge except that in line 3, $L(y)$ is replaced by $R(y)$.

Algorithm 1 IsLeftChildBackEdge(y, h).

```
1 flag ← False, w ← s;
2 for count < h do
3 |   if w == L(y) then
4 |   |   flag ← True;
5 |   w ← a neighbor of w guessed nondeterministically;
6 if w == y then
7 |   Return flag;
8 else
9 |   Abort;
```

▶ **Proposition 14.** *Given a graph G, vertex y and its height h with $G_h(s)$ being a simple path graph, if $(y, L(y))$ is a back edge in the subgraph $G_h(s)$ then IsLeftChildBackEdge returns True or Aborts. Conversely, if IsLeftChildBackEdge returns True, then $(y, L(y))$ is a back edge in the subgraph $G_h(s)$. Moreover, IsLeftChildBackEdge runs unambiguously.*

Proof. We start by arguing the correctness of IsLeftChildBackEdge. The argument for the correctness of IsRightChildBackEdge is similar.

Suppose $(y, L(y))$ is a back edge in the subgraph $G_h(s)$. Then there exists a path from s to y of length h such that $L(y)$ will be visited before y is reached. The lines 2-5 precisely does this check, and the flag will be set to True in that nondeterministic choice. Hence, the procedure returns True if the path ends in y. If the path does not end in y, since h is the length of the shortest path from s to y, it must be that the nondeterministic choices took at least one back edge during the traversal. In such cases, the procedure aborts in line 9.

We now argue that if IsLeftChildBackEdge returns True, then $(y, L(y))$ is a back edge in the subgraph $G_h(s)$. To argue this, suppose $(y, L(y))$ is not a back edge. Then, in any path from s to y, $L(y)$ will not be visited. Hence, in lines 2-5, the flag is never set to True in all nondeterministic choices. Hence, the procedure returns False if the path ends in y. If the path does not end in y, since h is the length of the shortest path from s to y, it must be that the nondeterministic choices took at least one back edge during the traversal. In such cases, the procedure aborts in line 9.

Unambiguity of IsLeftChildBackEdge. We argue the unambiguity of the procedure when it returns True or False. We recall that $G_h(s)$ is already a simple path graph and h is the length of the shortest path from s to y. We start with the case when the function returns True. Suppose the IsLeftChildBackEdge on (y, h) returns True for two nondeterministic choices. This corresponds to two walks from s to y. We claim that both of them must be *paths* of length h. To see this, suppose one of them is a walk of length h where a vertex v is repeated (such a v must exist). Then we can obtain a path of length strictly smaller than h by removing the vertices that appear between two consecutive v. We repeat this for every such v. This shows that y is at a distance strictly smaller than h, which is a contradiction.

With two paths of length h from s to y, the graph is no longer a simple path graph, which contradicts that $G_h(s)$ is a simple path graph. The argument for the unambiguity of IsRightChildBackEdge returning True is similar.

Now, suppose that the procedure returns False. The same argument holds verbatim with "True" replaced by "False". Since the procedure returns True and False unambiguously, the procedure is in $\mathsf{UL} \cap \mathsf{coUL}$. ◀

▶ Remark 15. The above procedures are not reach unambiguous as it is possible to have multiple walks in the graph that reach the same vertex (different from y) and end up aborting ambiguously.

3.2 Main procedure (Algorithm 2): Correctness and Unambiguity

Algorithm 2 Main(G, s, t).

```
1  reached ← False, n ← |V(G)|;
2  for d ← 1 ... n − 2 do
3      (y, h) ← (s, 0);
4      repeat
5          if t ∈ {L(y), R(y)} then
6              reached ← True;
7          if h == d and y is internal then
8              (y′, h′) ← (s, 0);
9              repeat
10                 if y′ ∈ {L(y), R(y)} then
11                     if ¬(IsLeftChildBackEdge(y, h) or IsRightChildBackEdge(y, h)) then
12                         Reject;
13                 if h′ == d and y ≠ y′ and y′ is internal and
                    {L(y′), R(y′)} ∩ {L(y), R(y)} ≠ ∅ then
14                     Reject;
15                 (y′, h′) = Next(y′, h′, d);
16             until (y′, h′) = (s, 0);
17         (y, h) = Next(y, h, d);
18     until (y, h) = (s, 0);
19 if reached then
20     Accept;
21 else
22     Reject;
```

In this section, we describe the procedure Main. We start with an overview, followed by a detailed description. This procedure takes in a graph G, two vertices, s and t, and accepts if and only if G is a simple path graph and there is a unique path from s to t in G. For each value of d, this procedure proceeds by verifying that up to a depth d, the given graph G is a simple path graph.

During this traversal, for each value of the depth, two pointers (y and y') are maintained. The procedure simultaneously checks two properties. First, it checks whether t appears as a neighbor of any visited vertex. Second, for each y at height d, it does an inner traversal using a second pointer y' to verify that G does not violate the conditions for a simple path graph for $G_d(s)$. Specifically, the inner traversal checks that (y, y') is not a cross edge or

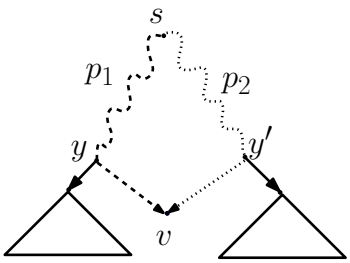

Figure 4 p_1 (shown as dashed), p_2 (shown as dotted) are of same length.

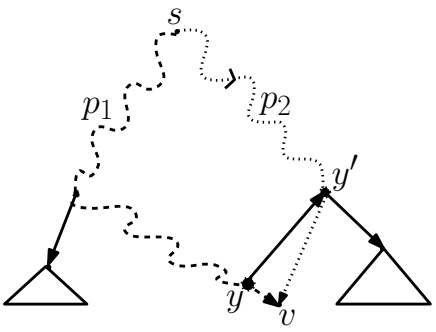

Figure 5 p_1 (shown in dashed), p_2 (shown in dotted) are of different length.

a forward edge, and no two internal vertices at depth d share a common neighbor. After we traverse the whole of the graph and all depth values have been processed, the algorithm accepts if and only if *reached* is True.

For the traversal, this procedure invokes Next that takes in a vertex y, its height h, and a depth d and returns the next vertex after y in the pre-order traversal of $G_d(s)$. The details of Next as well as the correctness and unambiguity will be explained in the next section (Section 3.3). The description of Main is given as Algorithm 2.

We argue its correctness.

▶ **Proposition 16.** *Given a graph G and two distinct vertices s and t in $V(G)$, $(G, s, t) \in$* SimplePath-Reach *iff* Main *accepts. Moreover,* Main *runs unambiguously in Accept and Reject.*

Proof. Suppose that $(G, s, t) \notin$ SimplePath-Reach. Then, it could be that either G is not a simple path graph or G is a simple path graph but there is no s to t path. We start with the former case.

Let p_1 and p_2 be two different simple paths from s to a vertex v in a graph G. In the figures that follow (Figures 4 and 5), they are dashed and dotted respectively. Without loss of generality, let v be the first point of convergence of p_1 and p_2. Let y appear on p_1 as a neighbor of v and y' appear on p_2 as a neighbor of v. The paths p_1 and p_2 can either be of the same length or of different lengths. On one hand, if they are of the same length (see Figure 4), either (y, v) or (y', v) forms a cross edge and the graph is rejected in line 14.

On the other hand, if the paths are of different lengths, without loss of generality, say y' has less height than y. Suppose, for a contradiction, the algorithm accepts. Then, the "Reject" at line 12 was never executed. We do not consider the "Reject" at line 14 since this is only executed when p_1 and p_2 are of same length.

There could be two reasons why the"Reject" at line 12 was never executed. (1) (y, y') is a back edge and y' is a child of y, and (2) y' is not a child of y.

For case (1), if (y, y') is a back edge, the path p_1 and p_2 meet at y' for the first time and not v (see Figure 5). This is a contradiction as y' is different from v. Then the algorithm must have rejected in this case. Now, for case (2), if y' is not a child of y, we have to show that the algorithm rejects. Since the inner loop compares y with every y' of height $h' \leq$ height of y, for some y, y' becomes v and y is the parent of y'. Now, (y, y') is not a tree edge (since y and y' appear on different paths) and not a cross edge (which would lead to reject at line 12), it must be a back edge, leading to a "Reject" in line 12.

Now, consider the case where G is a simple path graph, but there is no s to t path. Observe that the algorithm visits every vertex reachable from s in G. This is because the Next procedure is guaranteed to find the pre-order successor of the current vertex. Hence, as t was not reachable, the algorithm would reject.

Suppose $(G, s, t) \in$ SimplePath-Reach. Then, G is a simple path graph and there is an s to t path. In this case, none of the reject checks succeeds, and for some vertex y reachable from s, the vertex t will be a child of y. The algorithm is guaranteed to find such a y because Next gives the pre-order successor of the current vertex every time it is called. Consequently, *reached* is set to True in line 6. Hence, the algorithm accepts. It can be seen that Main is unambiguous. We give a short proof as follows.

Unambiguity of Main. The main procedure does not use any nondeterminism directly and uses nondeterminism only in the subroutines. Observe that all the subroutines (IsLeftChildBackEdge and IsRightChildBackEdge) are unambiguous in both "yes" and "no" instances by Proposition 14. Moreover, Next returns the pre-order successor of y and its height in the subgraph $G_d(s)$ unambiguously by Proposition 17. Hence, in Main, there cannot be two nondeterministic choices that end in Accept. Same holds for Reject. ◀

Proof of Theorem 1. To show that SimplePath-Reach $\in$ UL$\cap$coUL, we see that Main accepts SimplePath-Reach and the accept and reject are unambiguous by Proposition 16. This completes the proof. ◀

3.3 Next procedure (Algorithm 3): Correctness and Unambiguity

In this section, we describe the procedure Next as Algorithm 3. This procedure takes in a vertex y and two numbers h and d where it is guaranteed that the height of y in the subgraph $G_d(s)$ is h. This procedure returns the pre-order successor of y in the subgraph $G_d(s)$. The procedure assumes that h is the correct height of y and $G_d(s)$ is a simple path graph.

Now, let us briefly see how Next returns a vertex. There are three cases for a vertex y at height $h < d$ in Figures 6–8 and a case when height $h = d$ in Figure 9. In all these figures, the vertex highlighted with a gray fill and black border denotes the current vertex y, while the vertex with a white fill and black border denotes the vertex returned by Next(y, h, d). The shaded region indicates vertices that are unreachable from s. We now describe the cases.

Case 1: Left child of y is not a back edge. In this case, $L(y)$ is returned as the pre-order successor (Figure 6).

Case 2: Left child y is a back edge, but the right child is not. In this case, $R(y)$ is returned as the pre-order successor (Figure 7).

Case 3: Both children of y are back edges. Then, pre-order successor of y in $G_d(s)$ is returned (Figure 8).

Case 4: Now, suppose y is at height $h = d$. Even if y has children that are at a greater depth than d, Next does not return it. Instead Next returns the pre-order successor of y in $G_d(s)$ (Figure 9).

Illustrations of cases handled by Next

Procedure Next is described formally as Algorithm 3. Below, we argue its correctness.

▶ **Proposition 17.** *Given a graph G, vertex y, its height h and depth d of the subgraph $G_d(s)$, Next returns the pre-order successor of y in the graph $G_d(s)$ and its height. Moreover, Next returns y and its height unambiguously or aborts.*

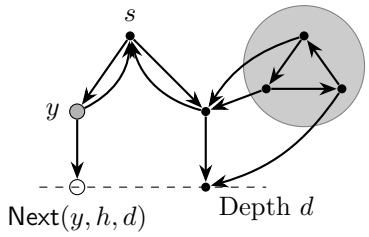

Figure 6 Case 1: When left child of y is not a back edge.

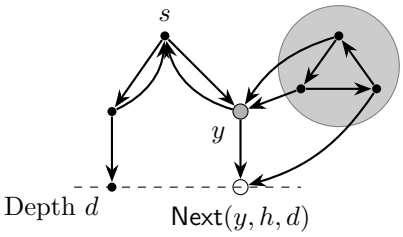

Figure 7 Case 2: When left child of y is a back edge, but the right child is not.

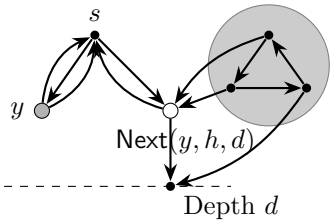

Figure 8 Case 3: When both children of y are back edges.

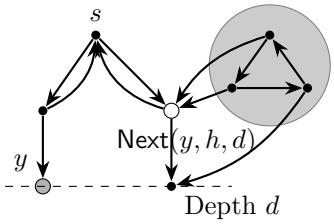

Figure 9 y is at height $h = d$.

Proof. The purpose of the procedure Next(y, h, d) is to return the pre-order successor of y in the graph $G_d(s)$. Below, we show that this indeed is the case. The pre-order successor of y can appear as a left child of y if the left child is not a back edge. If it is a back edge, the pre-order successor will be the right child of y, provided $(y, R(y))$ is not a back edge. If both left and right child of y are back edges, then the pre-order successor is not immediately clear as it needs to be computed. This is achieved using a call to PathUncle.

Suppose the next vertex in the pre-order traversal is the left child and $(y, L(y))$ is not a back edge. In this case, the procedure returns the left child and terminates. Suppose the next vertex in the pre-order traversal is the right child and $(y, R(y))$ is not a back edge. In this case, the procedure returns the right child and terminates. If the current vertex is the final vertex in the pre-order traversal of the graph $G_d(s)$, we return the source and terminate the procedure. Computing this final vertex is achieved in lines 10 to 18. If y is not the final vertex, we call PathUncle to return the next vertex. It can be seen that Next is unambiguous as shown below.

Unambiguity of Next. The procedure Next is unambiguous since it contains no nondeterministic statements except within the subroutines PathUncle, IsLeftChildBackEdge, and IsRightChildBackEdge. The subroutine IsLeftChildBackEdge and IsRightChildBackEdge are unambiguous in both "yes" and "no" instances by Proposition 14. Path uncle unambiguously returns the correct uncle of x along with its height by Proposition 18. Hence, Next is unambiguously returns a value or aborts. ◀

We remark that the only difference with the Next procedure appearing in Lange [13] is that in our case, (1) we need to check for back edges (Lemma 10) and (2) the "right most" vertex in $G_d(s)$ need not be the last vertex in the pre-order traversal; this needs to be computed.

Algorithm 3 Next(y, h, d).

```
1  if y is internal and
     ¬IsLeftChildBackEdge(y, h) and h < d
     then
2      Return (L(y), h + 1);
3  else
4      if y is internal and
         ¬IsRightChildBackEdge(y, h) and
         h < d then
5          Return (R(y), h + 1);
6      else
7          h' ← 0;
8          w ← s;
9          flag ← False;
10         while w is internal and h' ≤ d and
             flag ==False do
11             if ¬IsRightChildBackEdge(w, h)
                 then
12                 w ← R(w);
13                 h' ← h' + 1;
14             else if
                 ¬IsLeftChildBackEdge(w, h)
                 then
15                 w ← L(w);
16                 h' ← h' + 1;
17             else
18                 flag ← True;
19         if y == w then
20             Return (s, 0);
21         else
22             (u, h_u) = PathUncle(y, h);
23             Return (u, h_u)
```

Algorithm 4 PathUncle(x, g).

```
1  u ← R(s), w ← s;
2  count ← 1, h_u ← 1, h_w ← 0;
3  while count ≤ g and w is an internal
     vertex do
4      Guess b ∈ {0, 1};
5      if b == 0 then
6          if ¬IsLeftChildBackEdge(w, h_w)
             then
7              if
                 ¬IsRightChildBackEdge(w, h_w)
                 then
8                  u ← R(w);
9                  h_u ← h_w + 1;
10             w ← L(w);
11             h_w ← h_w + 1;
12     else
13         if ¬IsRightChildBackEdge(w, h_w)
             then
14             w ← R(w);
15             h_w ← h_w + 1;
16     count ← count + 1;
17 if w == x then
18     Return (u, h_u);
19 else
20     Abort;
```

Figure 10 Algorithms Next and PathUncle.

3.4 PathUncle (Algorithm 4): Correctness and Unambiguity

This procedure is executed as a subroutine of the previously defined algorithm Next. Given a vertex x and its height g, the algorithm PathUncle(x, g) unambiguously returns the correct uncle of x along with the uncle's height. This is achieved by guessing a path from the root s to x, at each step storing the latest encountered potential uncle u (i.e., the right child of the current node) and its corresponding height h_u. To ensure guesses form a valid path, the algorithm avoids traversing edges that are identified as back edges, using the procedures IsLeftChildBackEdge and IsRightChildBackEdge. The algorithm verifies the correctness of its guesses by checking that the guessed path ends at x. If the path is invalid, the algorithm aborts. Otherwise, it returns the uncle that is farthest from s along the simple path to x.

▶ **Proposition 18.** *Given a vertex x and its height g, the algorithm PathUncle(x, g) unambiguously returns the correct uncle of x along with the uncle's height or aborts.*

Proof. We argue the correctness of PathUncle. Note that there is exactly one path $P_{s,x}$ from s to x as x is reachable from s and $G_d(s)$ is a simple path graph. Observe that the while loop (lines 3 to 16) eventually terminates because, in each iteration, the value of *count* is incremented irrespective of the guess taken. We show that the uncle u must appear as a right child of one of the vertices in $P_{s,x}$. To argue this, we claim the following "(1) Uncle of w is u with h_u as its height (2) h_w is the height of w" is maintained at the end of line 16. Item (2) is necessary for ensuring correctness of call to IsLeftChildBackEdge and IsRightChildBackEdge functions.

We argue the invariant by induction on *count*. For $count = 1$, $w = s$ and the uncle u of w is $R(s)$. Moreover, $h_u = 1$ and $h_w = 0$ are the correct heights of u and w respectively. Hence, the base case holds. Moving to the induction case, let $count = c$ for some $c \geq 1$ and w be an internal vertex reached. By induction hypothesis, the uncle of w is u, and h_u is its height, and h_w is the height of w. We make the following claim.

▶ **Proposition 19.** *Let G be a simple path graph and w be a vertex such that $(w, R(w))$ is a back edge in G. Then, the uncle of w is the right child of the unique vertex v in the unique path $P_{s,w}$ such that (a) v has a right child that is not a back edge and, (b) Every vertex r from v to w in $P_{s,w}$ the edge $(r, R(r))$ is either an edge in $P_{s,w}$ or is a back edge.*

In the induction step, w gets updated to $R(w)$ or $L(w)$ based on the nondeterministic choice. We now argue that for both cases, h_w gets updated correctly. This would prove (2) from the induction. In both cases, provided $L(w)$ or $R(w)$ is *not* a back edge, w is updated appropriately, and h_w is incremented by 1. By induction, h_w had the correct height of w. Hence, incrementing h_w gives the correct height of the new w.

We now argue (1) from induction. Suppose that the nondeterministic choice was to the right (that is, $b = 1$) and the right child is verified to be not a back edge. In this case, u and h_u stay the same. Hence, the uncle of $R(w)$ is the same as that of the uncle of w, which is u. Suppose that the nondeterministic choice was to the left (that is, $b = 0$) and the left child is verified to be not a back edge. Now, there are two cases. *Case 1*: The right child of w is a back edge. In this case, the uncle and its height are not updated. With $(w, R(w))$ being a back edge, from Proposition 19, we conclude that the uncle of $L(w)$ is the same as that of uncle of w. By induction hypothesis, with the uncle of w being u, the uncle of $L(w)$ is also u, and height of u is the same as it is not updated. *Case 2*: Right child of w is not a back edge. In this case, in step 8, the uncle is updated to $R(w)$. This is correct since the conditions of Proposition 19 are satisfied. Since u is updated to be the right child of w, the height of u is one more than the height of w. Hence, by induction, irrespective of the nondeterministic choice, the uncle of w is always u. For the unique nondeterministic choice reaching x, u holds the uncle of x. In the case where the verification steps (back edge checks) fail, the *count* gets incremented and becomes equal to g prematurely. Since g is the length of the shortest path to x, w never becomes x, and the procedure aborts.

Unambiguity of PathUncle. Towards this, we argue that for the execution of the procedure in which all nondeterministic choices are correct, the procedure returns the correct uncle and its associated height. On the other hand, for any execution path involving an incorrect nondeterministic choice, the procedure aborts.

We first argue the case when all the nondeterministic choices taken by the algorithm are correct. We show that a unique (u, h_u) is returned. This is true since for every w, there is a unique path $P_{s,w}$ from s to w in $G_d(s)$ and there is a unique vertex v in the path $P_{s,w}$ whose right child is the uncle. Hence, the returned uncle is unique. Now consider the case in which the procedure makes an incorrect nondeterministic guess, which can occur in one of

two ways: (1) the path of length g chosen by the procedure starting from vertex s does not terminate at vertex x, or (2) the path length is less than g because w reached a leaf vertex in $G_d(s)$. In both cases, the path terminates at vertex w, different from x as g is the length of the shortest path from s. Here, the procedure aborts in line 20. These aborts need not be unique as there could be multiple nondeterministic choices that could end in the same w. ◀

Proof of Proposition 19. Observe that such a v satisfying (a) and (b) always exists. This is because the right child of s is a candidate uncle for any vertex in $P_{s,w}$. The uniqueness follows since the pre-order traversal is unique.

Suppose, for contradiction, the right child of v is not the correct uncle of w but satisfies conditions (a) and (b). Observe that the uncle of w cannot appear as a right child of any vertex in the path $P_{s,v}$. Hence, the uncle of w must appear as a right child of a vertex r between v and w in $P_{s,w}$. However, for any r, $(r, R(r))$ is neither an edge in $P_{s,w}$ nor a back edge. This violates (b). ◀

Having established that reachability in simple path graphs is in $\mathsf{UL} \cap \mathsf{coUL}$, we now turn to a broader question about the complexity of recognition versus reachability in various special classes of graphs in Section 4.

4 Recognition versus Reachability for reach unambiguous graphs

Let $\mathcal{G}$ be a family of graphs. Recall the recognition ($\mathcal{G}$-Rec) and reachability ($\mathcal{G}$-Reach) problems from preliminaries. Clearly, solving the reachability problem suffices to solve the recognition problem as well. This means $\mathcal{G}\text{-Rec} \leq_m^{log} \mathcal{G}\text{-Reach}$, showing that solving reachability is as "hard" as recognition. Since $\mathcal{G}$-Reach where $\mathcal{G}$ is reach unambiguous graphs, is in ReachUL [13], the recognition problem is also solvable in ReachUL by the aforementioned observation. We now show that the reachability problem also reduces to the recognition problem for the family of reach unambiguous graphs. Since reachability for reach unambiguous class is ReachUL-complete, our result shows that the recognition problem is also ReachUL-complete (under many-one logspace reductions). We now prove Proposition 2.

Proof of Proposition 2. The reach unambiguous machine M' with oracle access to $L(M)$ for $\mathcal{G}$-Reach is as follows. Given (G, s, t), first check if $G \in \mathcal{G}$-Rec by making an oracle query to $L(M)$. If $G \notin \mathcal{G}$-Rec, then reject. Otherwise, $G \in \mathcal{G}$-Rec, and guess a length $\ell <\mid V(G) \mid$. Now, guess a path of length ℓ from s to t. If the path ends in x where $x \neq t$, abort with (ℓ, x). On the other hand, if $x = t$, then accept.

We show that $L(M') = \mathcal{G}$-Reach. Suppose M' accepts (G, s, t), then it must be that $G \in \mathcal{G}$-Rec, and there is a path from s to t in G. Hence, $(G, s, t) \in \mathcal{G}$-Reach. Suppose $(G, s, t) \in \mathcal{G}$-Reach, then $G \in \mathcal{G}$-Rec and there is a path from s to t in G. Hence, the oracle call to $L(M)$ succeeds. Moreover, there is exactly one guess for length $\ell < |V(G)|$ and exactly one path starting from s that ends in t. Hence, the machine M' accepts (G, s, t).

We now argue why M' is reach unambiguous. Consider the case where $\mathcal{G}$ is reach unambiguous graphs. Suppose $(G, s, t) \notin \mathcal{G}$-Reach. Either $G \notin \mathcal{G}$-Rec or there is no s to t path. Note that the case where there is more than one path from s to t is covered in the $G \notin \mathcal{G}$-Rec. Suppose $G \notin \mathcal{G}$-Rec. Then the oracle will return a "no" answer and M' rejects unambiguously. Now, suppose $G \in \mathcal{G}$-Rec, and there is no path from s to t in G. Then (ℓ, x) must be unique for all nondeterministic choices made by M'. Hence, M' is reach unambiguous.

Since $\mathsf{ReachUL}^{\mathsf{ReachUL}} = \mathsf{ReachUL}$ by Proposition 3, $L(M') = \mathcal{G}$-Reach is in ReachUL. ◀

5 Summary and Open Problem

The reachability problem restricted to reach unambiguous graphs is complete for the class ReachUL (under many-one logspace reductions) [13]. In this work, we studied the complexity of reachability problems specifically for simple path graphs (a superclass of reach unambiguous graphs) and showed that the problem is in $\mathsf{UL} \cap \mathsf{coUL}$. Since $\mathsf{ReachUL} \subseteq \mathsf{UL} \cap \mathsf{coUL}$, as an open question, we ask whether our result can be improved to ReachUL.

Open Question. Is the complexity of the reachability problem for simple path graphs, SimplePath-Reach $\in$ ReachUL?

Such an improvement would give a simultaneous $\mathsf{poly}(n)$ time and $O(\log^2 n)$ space algorithm for this problem, placing it in SC^2 and thereby improving [11].

References

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

2 Eric Allender and K-J Lange. $\mathsf{RUSPACE}(\log n) \subseteq \mathsf{DSPACE}(\log^2 n/\log\log n)$. *Theory of Computing Systems*, 31(5):539–550, 1998.

3 Carme Àlvarez and Raymond Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, 9:123–145, December 2000. doi:10.1007/PL00001603.

4 Simon Apers and Roman Edenhofer. Directed st-Connectivity with Few Paths Is in Quantum Logspace. In *40th Computational Complexity Conference (CCC 2025)*, volume 339 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.CCC.2025.18.

5 S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

6 Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. *ACM Trans. Comput. Theory*, 1(1), February 2009. doi:10.1145/1490270.1490274.

7 Gerhard Buntrock, Birgit Jenner, Klaus-Jörn Lange, and Peter Rossmanith. Unambiguity and fewness for logarithmic space. In *Proceedings of the 8th International Symposium on Fundamentals of Computation Theory*, FCT '91, pages 168–179, Berlin, Heidelberg, 1991. Springer-Verlag. doi:10.1007/3-540-54458-5_61.

8 Stephen A. Cook. Deterministic cfl's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 338–345. Association for Computing Machinery, 1979. doi:10.1145/800135.804426.

9 T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, third edition*. Computer science. MIT Press, 2009. URL: https://books.google.co.in/books?id=i-bUBQAAQBAJ.

10 Chetan Gupta, Vimal Raj Sharma, and Raghunath Tewari. Reachability in O(log n) Genus Graphs is in Unambiguous Logspace. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126, pages 34:1–34:13, 2019. doi:10.4230/LIPIcs.STACS.2019.34.

11 Sampath Kannan, Sanjeev Khanna, and Sudeepa Roy. STCON in Directed Unique-Path Graphs. In Ramesh Hariharan, Madhavan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2, pages 256–267, Dagstuhl, Germany, 2008. doi:10.4230/LIPIcs.FSTTCS.2008.1758.

12 Dexter Kozen. *Theory of Computation*. Texts in Computer Science. Springer, 2006. doi:10.1007/1-84628-477-5.

13 Klaus-Jörn Lange. An unambiguous class possessing a complete set. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 339–350. Springer, 1997. doi:10.1007/BFB0023471.

14 Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):1–24, 2008. doi:10.1145/1391289.1391291.

15 Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118–1131, 2000. doi:10.1137/S0097539798339041.

16 Avi Wigderson. The complexity of graph connectivity. In *International Symposium on Mathematical Foundations of Computer Science*, pages 112–132. Springer, 1992. doi:10.1007/3-540-55808-X_10.