# A log-space algorithm for reachability in planar DAGs with few sources*

Derrick Stolee†+ Chris Bourke+ N. V. Vinodchandran+

+Department of Computer Science & Engineering
†Department of Mathematics
University of Nebraska–Lincoln
Lincoln, NE 68588-0115
{dstolee,cbourke,vinod}@cse.unl.edu

April 15, 2009

**Abstract**

Designing algorithms that use logarithmic space for graph reachability problems is fundamental to complexity theory. It is well known that for general directed graphs this problem is equivalent to the NL vs L problem. For planar graphs, the question is not settled. Showing that the planar reachability problem is NL-complete would show that nondeterministic log-space computations can be made unambiguous. On the other hand, very little is known about classes of planar graphs that admit log-space algorithms. We make progress in this direction. We show that reachability in planar DAGs with $O(\log n)$ number of sources can be solved in log-space. We use a new decomposition technique for planar DAGs as a basis for our algorithm.

*This work was supported by the NSF grant CCF-0430991.

# 1 Introduction

Graph reachability problems are central to complexity theory. The $st$-reachability problem of deciding whether there exists a path from a node $s$ to a node $t$ in a directed graph is complete for nondeterministic log-space (NL). Very recently, in a break-through result, Reingold showed that $st$-reachability problem over undirected graphs is complete for deterministic log-space (L) [11]. Various versions of reachability problem characterize various complexity classes within NL [3,4,7,11]. Because of its central role in complexity theory, designing algorithms that use logarithmic space for graph reachability problems is a fundamental question.

Planarity has proven to be a very important restriction when dealing with graph problems, both theoretically and algorithmically. Algorithmically, because of certain fundamental structural theorems such as the Lipton-Tarjan planar separator theorem [10], many computational problems over planar graphs admit algorithms with better running time and/or parallelism. However, from a space-complexity view point progress has started to emerge only recently [2, 6, 11]. In particular, the space complexity of $st$-reachability problem over planar graphs currently is far from being completely settled. It is known to be hard (under projection reductions) for deterministic log-space [7], but not known to be complete for NL. Recently, it was shown that this problem can be solved in unambiguous logarithmic space (the class UL) [5]. Hence if reachability for planar graphs is complete for NL then all of nondeterministic log-space computation can be made unambiguous (that is NL = UL). While this is very likely, proving NL = UL will be a major result in complexity theory. On the other hand, very little is known about classes of planar graphs that admit log-space algorithms. Jacoby et al. show that various reachability and optimization questions for *series-parallel* graphs admit deterministic log-space algorithms [8, 9]. Series-parallel graphs are a very restricted subclass of planar directed acyclic graphs (DAGs). In particular, such graphs have a single source and a single sink (single source single sink DAGs are sometimes called $st$-graphs in the literature). Recently, Allender et al. [1] extended Jacoby et al.'s result to show that $st$-reachability for planar DAGs with single source and multiple sinks can be decided in log-space. Using a reduction, the authors were able to slightly improve this upper bound to planar DAGs with *two* sources and multiple sinks. This remains the current best class of planar DAGs that admit deterministic log-space algorithms.

In this paper, we make moderate progress on this situation. We show that reachability in planar DAGs with $O(\log n)$ number of sources and multiple sinks can be solved in deterministic log-space.

**Theorem 1.1.** *Reachability in planar directed acyclic graphs with $O(\log n)$ sources can be decided in deterministic log-space.*

A few words about our technique: our algorithm builds on the aforementioned Allender et al.'s log-space algorithm for single source multiple sink planar DAGs (denoted as SMPD in [1]). Our first step is to decompose the graph with multiple sources into "SMPD components" in a simple manner. Each component allows for reachability testing, but to manage the interaction of these components we present a new technique of classifying edges into topological equivalence classes, bringing a previously unused property of planar graphs into the algorithmic realm.

# 2 Preliminaries

Let $G = (V, E)$ be a simple acyclic planar digraph with $m$ sources. By the results of Allender & Mahajan [2] and subsequently Reingold [11], we can compute the combinatorial embedding of $G$ in

log-space. We delay the question of recognizing acyclic graphs and will assume the input graph is a DAG until this problem is resolved in Section 4. We wish to solve reachability for given vertices $u$ and $v$. That is, we want to determine if there exists a directed path $u \rightsquigarrow v$.

$G$ is a Single-source Multiple-sink Planar DAG (SMPD) if it has only a single source and no restriction on the number of sinks. Allender et al. [1] showed that reachability in SMPDs can be decided in deterministic log-space. Thus, if $m = 1$, we can appeal to their algorithm to solve reachability.

Our algorithm will work for any number of sinks $m$, but for graphs with $m = O(\log n)$ sources it will only use a logarithmic amount of space. We call such graphs Log-source Multiple-sink Planar DAGs (LMPDs).

Since $G$ is acyclic, without loss of generality, we can assume that $u$ is a source and $v$ is a sink by deleting incoming and outgoing edges respectively without affecting reachability. For each non-source vertex $x$, choose an arbitrary incoming edge $e$. Since a single incoming edge is chosen per vertex, and $G$ has no directed cycles, the subgraph corresponding to these edges is a forest with $m$ directed *source trees* each rooted at a source. We depict each tree $T_i$ as organized radially around its source $s_i$, for $i = 1, \ldots, m$. Figures depict these trees as circles with leaves on the border and the source label in the center (cf. Figure 1). We consider $v$ to be a component on its own, with no special incoming edge. Since $u$ is assumed to be source, it is the root of a tree, $T_u$, as well.

The construction of the forest naturally partitions edges into one of the following five types. There are two types of edges that are easy to define.

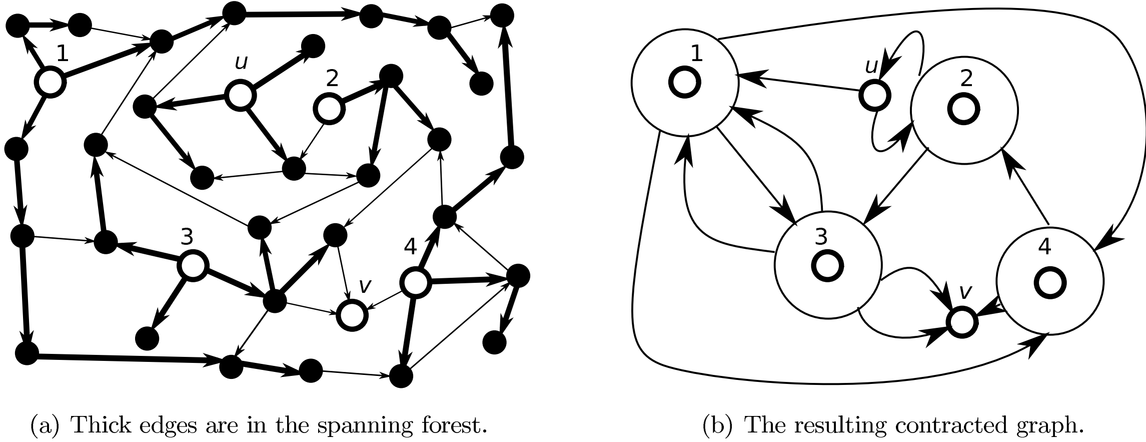
- A *tree* edge is an edge in the forest.
- A *launch* edge is an edge between vertices in different trees.

Now consider an edge $e = (x, y)$ with $x$ and $y$ in the same tree, $T_i$, but is not a tree edge. Let $\text{LCA}(x, y)$ denote the least common ancestor of nodes $x, y$ in the tree. That is, the vertex farthest from the root that has $x$ and $y$ as descendants. A closed curve is defined in the underlying undirected planar embedding by the paths from $x$ to $\text{LCA}(x, y)$ to $y$ and the edge $e$. Call this curve the *tree-closing curve*. The other edge types are defined using this curve, by counting the number of vertices from $T_i$ that are within the curve or outside the curve. If one of the counts is zero, we say this partitions the vertices trivially. This process of counting avoids issues with which face is declared to be the outer face and we can consider the embedding to be on a sphere. These new edge types follow:

- A *local* edge trivially partitions the vertices not on the curve.
- A *jump* edge closes a curve that does not trivially partition the vertices, but does trivially partition the sources $s_1, s_2, \ldots, s_m, u$, and $v$.
- A *loop* is an edge such that the curve does not trivially partition the sources.

Given an edge and a planar combinatorial embedding it is easy to determine in $\mathsf{L}$ which among the five types of edges it is. Moreover, each tree $T_i$ along with its tree, local, and jump edges constitutes an SMPD and so reachability questions for vertices within a single tree can also be answered in deterministic log-space [1].

For the sake of understanding the interactions between these SMPD components, imagine contracting all tree edges in $G$ so that the only vertices remaining are $u, v, s_1, \ldots, s_m$. Then, consider only the launch and loop edges in this contraction. This forms a planar multigraph $H$ with $m + 2$ vertices that may require the edges to be drawn as curves and not straight lines. For an example of such a contraction, see Figure 1.




(a) Thick edges are in the spanning forest.

(b) The resulting contracted graph.

Figure 1: A planar DAG contracting to a planar multigraph.

We will consider reachability at each source tree to be a subproblem discussed in the next section. Then, we develop a method of classifying the launch and loop edges into $O(m)$ classes in Section 2.2, forming the basis of our algorithm.

## 2.1 Exploring SMPDs

It will be useful to recall some high-level details of the algorithm of Allender et al. [1] for reachability in SMPDs. In fact, we must present the algorithm again to use specific details of the computation in our later extensions. For an SMPD $G$, there is a path from the single source $s$ to *every* vertex. Thus, the task is to determine if there exists a path between two arbitrary vertices $u, v$.

**Theorem 2.1** (Allender et al. [1]). *Given an SMPD $G$ and two vertices $u, v$, determining if there exists a path $u \rightsquigarrow v$ is decidable in deterministic log-space.*

In our case, using SMPD as a subproblem does not actually look for individual vertices that can be reached, but instead focus on expanding an *explored region* around a vertex $x$. We want to define a region that describes a reachable set of vertices and edges with a constant number of variables. Call the algorithm SMPDExplore($x$) to explore around a vertex $x$ on its source tree.

An important observation of Allender et al. [1] is that reachability using only tree and local edges is easily decidable in $\mathsf{L}$: remove all jump edges, and all leaves are on the external face. Connect these leaves to a new vertex and the resulting graph is a Single-source Single-sink Planar DAG (SSPD) which they show is computable in $\mathsf{L}$. Denote the set of vertices reachable from a vertex $x$ using tree and local edges as ReachLocal($x$).

Define a vertical axis as follows: $s$ is in the center, the tree path from $s \rightsquigarrow x$ is oriented above $s$. This axis naturally splits the graph into left and right directions from $x$, with all other vertices and edges oriented radially around $s$ in either an outward, clockwise or counterclockwise manner. This defines a predicate: IsClockwise($y, w, z$) which is True if the vertices $y, w$, and $z$ appear in clockwise order around the source $s$, or if any pair are ancestor/descendant in the source tree. This can be computed in log-space using a fast computation involving LCA($x, y$) in several cases.

For all vertices within ReachLocal($x$), define the vertex that is farthest in the clockwise direction to be LocalRight($x$). That is, LocalRight($x$) $= y$ if $y \in$ ReachLocal($x$) and for all

$z \in \text{ReachLocal}(x)$, either $\text{IsClockwise}(x, z, y)$ or $\text{IsClockwise}(z, x, y)$. Break ties by choosing the vertex closest to the source. The vertex farthest in the counter-clockwise directions is $\text{LocalLeft}(x)$.

SMPDExplore defines an *explored region* by two variables $\mathsf{JumpLeft}$ and $\mathsf{JumpRight}$. It will always be the case that $\text{IsClockwise}(\mathsf{JumpLeft}, x, \mathsf{JumpRight}) = \textsc{True}$, except for a minor case that will be mentioned later. A vertex $y$ is in the explored region if $\text{IsClockwise}(\mathsf{JumpLeft}, y, \mathsf{JumpRight}) = \textsc{True}$ and $y$ is not an ancestor of $\mathsf{JumpLeft}$ or $\mathsf{JumpRight}$. We initialize $\mathsf{JumpLeft} = \text{LocalLeft}(x)$ and $\mathsf{JumpRight} = \text{LocalRight}(x)$.

Each iteration of the SMPD algorithm attempts to extend the explored region by "jumping" out of the explored region as little as possible. We say a jump edge *jumps out* of the explored region if its tail is in the explored region and its head is not. All jump edges that jump out of the explored region can be enumerated and we choose at most two edges, one for each direction. We choose the jump edge with head closest to $\mathsf{JumpLeft}$ in the counter-clockwise direction and the jump edge with head closest to $\mathsf{JumpRight}$ in the clockwise direction. Let the heads of these edges be $\ell$ and $r$, respectively. Enumerate $\text{ReachLocal}(\ell)$ and $\text{ReachLocal}(r)$ and set $\mathsf{JumpLeft} = \text{LocalLeft}(\ell)$ and $\mathsf{JumpRight} = \text{LocalRight}(\ell)$. Eventually, either there will be no edges jumping out of the explored region, or $\mathsf{JumpLeft}$ and $\mathsf{JumpRight}$ will overlap and $\text{IsClockwise}(\mathsf{JumpLeft}, x, \mathsf{JumpRight}) = \textsc{False}$. In this second case, we define the explored region to be the entire tree, as there are paths from $x$ going around the entire tree in each direction.

Now that the algorithm is described, there is an additional property that we require.

**Proposition 2.2.** *At every iteration of the algorithm for SMPD on a tree $T_i$ from a vertex $x$, if $y$ is in the explored region, but not reachable from $x$, then there are no jump edges from $y$ to a vertex outside the explored region. Moreover, there are no launch or loop edges starting at $y$.*

*Proof.* Note that for any $y$ in the initial explored region has one of two cases. Either $y$ is in $\text{ReachLocal}(x)$, or $y$ is an ancestor of a vertex $z$ in $\text{ReachLocal}(x)$, since there is a path from $x$ to each of $\mathsf{JumpLeft}$ and $\mathsf{JumpRight}$ using tree and local edges. Let $z = \text{LCA}(\mathsf{JumpLeft}, \mathsf{JumpRight})$, and $z$ is an ancestor of $y$, with the path to $y$ between the paths from $z$ to $\mathsf{JumpLeft}$ and $\mathsf{JumpRight}$. This implies that $y$ is properly contained within the closed curve starting at $z$ taking the tree path to $\mathsf{JumpLeft}$ then the tree and local path to $x$ then the tree and local path to $\mathsf{JumpRight}$ and finally the tree path to $z$. Thus, any edges from $y$ must stay within these bounds.

Proceed with iteration, assuming that the property holds at each step until the $i$th iteration. The initial explored region has already shown the result. Consider values $x, \mathsf{JumpLeft}, \mathsf{JumpRight}$ resulting from $i$ iterations of the algorithm. It is enough to show that the next iteration maintains the property.

Let $\ell$ and $r$ be the vertices reached by jumping out of the explored region. If a vertex $y$ would be in the new explored region, it would have $\text{IsClockwise}(\text{LocalLeft}(\ell), y, \text{LocalRight}(r)) = \textsc{True}$, and would not be an ancestor of $\text{LocalLeft}(\ell)$ or $\text{LocalRight}(r)$. If $y$ is in the old explored region, the result already holds. Thus, consider $y$ to be a vertex that is in the new explored region, but not the old explored region. That is, $\text{IsClockwise}(\text{LocalLeft}(\ell), y, \mathsf{JumpLeft}) = \textsc{True}$ or $\text{IsClockwise}(\mathsf{JumpRight}, y, \text{LocalRight}(r)) = \textsc{True}$.

There is nothing to prove if $y$ is reachable from $x$, so assume not. The vertices $\ell$ and $r$ are reachable from $x$, as the tails of the jump edges from the previous iteration are reachable from $x$. We must show that $y$ is within a closed curve blocking any edges incident to $y$ from reaching vertices outside the explored region.

Consider the case that $\text{IsClockwise}(\text{LocalLeft}(\ell), y, \mathsf{JumpLeft}) = \textsc{True}$. If it is also true that $\text{IsClockwise}(\text{LocalLeft}(\ell), y, \text{LocalRight}(\ell)) = \textsc{True}$, then the argument for the initial explored

region works for here, as if we began searching at $\ell$. Otherwise, we have $y$ in the region "skipped" by the jump edge. That is, IsClockwise(LocalRight($\ell$), $y$, $\mathsf{JumpLeft}$) = TRUE. But now consider the edge that jumps to $\ell$. The tail of this edge is a vertex $\ell'$ in the old explored region. The tree path from $\ell'$ to LCA($\ell', \ell$) to $\ell$ and then this jump edge closes a region containing $y$. Thus, edges leaving $y$ remain in this region, and cannot jump out of the explored region.

The same argument holds for the case where IsClockwise($\mathsf{JumpRight}$, $y$, LocalRight($r$)) = TRUE.

□

For use in the extended algorithm, the explored region around $x$ is recomputed using FarthestLeft($x$) = $\mathsf{JumpLeft}$ and FarthestRight($x$) = $\mathsf{JumpRight}$, returning the associated values from the algorithm.

## 2.2 Topological Equivalence of Edges

Consider the contracted graph $H$ as previously defined. This is a planar multigraph with curved edges. Any two edges with the same endpoints are compared by the closed curve they define in the plane. This divides the plane into three parts: the curve itself, and two disjoint regions. The two endpoints are within the curve, but the other vertices are in exactly one of these regions. This defines a partition of the other vertices into two parts. If one part is empty, then it can be imagined that these edges could be merged without crossing any vertices by traveling through this region. This is called a trivial partition and is the essential property that makes two edges equivalent for our purposes.

**Definition 2.3.** Two edges $e_1, e_2$ are *topologically equivalent* if they have the same end points, and the closed curve they create partitions the vertices into the two endpoints on the curve, the empty set, and all other vertices.

It is easy to see that the relation "$e_1$ is topologically equivalent to $e_2$" is actually an equivalence relation. Denote the equivalence class of an edge $e$ by $[e]$, but these classes will frequently be labeled with a letter, such as $C$ or $D$, when an edge is not provided. There are a few properties of how these classes appear in a combinatorial embedding that corresponds to a planar embedding.

**Lemma 2.4.** *All topologically equivalent edges (that are not loops) with an endpoint at a vertex $v$ appear consecutively when listing the edges incident to $v$ in a clockwise order.*

The definition of topological equivalence also works for loops by separating the incident vertex into two close vertices and using the edge definition. But, working with loops is slightly more complicated. We refer to a loop at a vertex $v$ as *trivial* if the region closed by the loop trivially partitions $V - \{v\}$. In the graph $G$, these would correspond to jump edges as they do not separate the sources, so they will not appear in $H$.

**Lemma 2.5.** *All topologically equivalent loops at a vertex $v$ appear in at most two blocks when listing the edges incident to $v$ in a clockwise order, excluding trivial loops.*

Given an arbitrary multigraph on $n$ vertices, the number of possible equivalence classes is finite, but large. For every pair of vertices $u, v \in V(G)$, we can define a surjective map from the possible equivalence classes between $u$ and $v$ to the subsets of $V(G) - \{u, v\}$, modulo the complement operation. Choose an arbitrary edge between $u$ and $v$. Any other edge forms a closed, but not necessarily simple, curve. This curve defines an interior and exterior, which partitions the vertices $V - \{u, v\}$ into two parts.

In terms of $k$, the number of equivalence classes is at least exponential. However, if we consider the size of the graph, its easy to see that the number of equivalence classes is bounded by $|E|$ and so is always polynomial in the size of the graph. Since $G$ is simple, each multi-edge in $H$ is derived from a launch or loop edge in $G$, so is not increasing the size.

Nevertheless, it is interesting to note that even for planar multigraphs, the number of equivalence classes is linear in $n$. By Euler's formula [12], the number of edges in a planar graph is bounded; $|E| \leq 3|V| - 6$. Surprisingly, an inductive counting argument gives the same bound on the number of equivalence classes for planar multigraphs.

**Theorem 2.6.** *A planar multigraph on $n$ vertices has at most $3n-6$ equivalence classes of edges and non-trivial loops.*

*Proof.* Let $M_n$ be the maximum number of edge classes present in a planar graph on $n$ vertices. Note that $M_1 = 0$, $M_2 = 1$, $M_3 = 3$. We proceed by induction to show that for $n \geq 4$, $M_n \leq M_{n-1} + 3$, giving $M_n \leq 3(n-3) + 3 = 3n - 6$.

Let $G$ be a planar multigraph of order $n$ with a maximal number of topological edge classes. We may assume there is exactly one edge of each class. Let $G'$ be the simple graph on the same vertices with the collapsed edge set; $u$ and $v$ are adjacent in $G'$ exactly when there is one or more edges in $G$ between them.

By the degree-sum formula and $|E| \leq 3|V| - 6$, there exists a vertex $v$ of degree at most five in $G'$. Thus, $k = |N_G(v)| \leq 5$. Arrange the adjacent vertices radially around $v$. Denote them $u_1, \ldots, u_k$ in clockwise order. We will use subscripts modulo $k$.

If $k = 1$, then $v$ is contained in a loop. This loop is trivial in $G - v$, and there is only one edge class from $v$ to $u_1$ that can be contained in this loop. Since $G - v$ has at most $M_{n-1}$ edge classes, and we add the loop and the edge to get $G$, we have at most $M_{n-1} + 2$ edges in $G$, which satisfies the theorem.

**Claim 2.1.** *There is a single edge class from $v$ to $u_i$ for each $i$. Furthermore, there are edge classes present for the edges $\{u_i, u_{i+1}\}$ that form a cycle separating $v$ from the vertices not in its neighborhood.*

It is helpful to note that there must be a closed curve formed by edges that separates $v$ from the other vertices or else we could add another edge and $G$ would not be maximal. Thus, there are edges between $u_1, \ldots, u_k$ that form a cycle, but are not necessarily in the cyclic order. Let $u_{\ell_1}, \ldots, u_{\ell_m}$ be the inner-most cycle of edges present that separates $v$ from the exterior vertices. Then, if $u_i$ is not in this cycle, there must be an edge $\{u_{\ell_j}, u_{\ell_{j+1}}\}$ that passes around it and $v$. But, we can add the edges $\{u_{\ell_j}, u_i\}$ and $\{u_i, u_{\ell_{j+1}}\}$ to form an inner cycle. This contradicts the maximality of $G$.

Finally, we have a cycle around the neighborhood of $v$. Thus, all edges from $v$ to $u_i$ must be in this region with $v$, and thus are topologically equivalent as they cannot properly partition the vertices.

Now, we have $v$ of degree at most five in $G$ as well, outside of loops. However, all loops at $v$ must be trivial and we omit them. By induction there are at most $M_{n-1}$ edges in $G - v$, and we add at most five edges to $G - v$ to form $G$. Thus, we have $M_n \leq |E(G - v)| + \deg(v)$.

If $\deg(v) \leq 3$, we have $M_n \leq |E(G - v)| + \deg(v) \leq M_{n-1} + 3$.

If $\deg(v) = 4$, we can add an chord to the cycle $u_1u_2u_3u_4$, thus $|E(G - v)| \leq M_{n-1} - 1$, and we have $M_n \leq M_{n-1} - 1 + 4 = M_{n-1} + 3$.

If $\deg(v) = 5$, we can add two chords to the cycle $u_1u_2u_3u_4u_5$, thus $|E(G - v)| \leq M_{n-1} - 2$, and we have $M_n \leq M_{n-1} - 2 + 5 = M_{n-1} + 3$. □

This gives $O(m)$ possible equivalence classes in $H$, which is a crucial bound in our result.

It is important to test if the edges of $H$ are topologically equivalent using only the information from $G$. Two launch edges $e_1$ and $e_2$ in $G$ have the same topological class in $H$ if

1. $e_1$ and $e_2$ connect the same trees $T_i$ and $T_j$ (i.e., have the same endpoints in $H$), and
2. the cycle formed by the tree paths from $s_i$ to $e_1$, $e_1$ to $s_j$, $s_j$ to $e_2$, and $e_2$ to $s_i$ trivially partition the vertices $u$ and $v$ and the other sources (i.e., the closed curve they form partitions the other vertices of $H$ trivially).

This process can be computed in log-space by enumerating all vertices and tracking a count of the important vertices in and outside of the cycle. Hence, we can enumerate all edges of the same class when given a representative edge.

**Remark.** The term *topological* appears in this definition, because the definition can really be interpreted using homeomorphisms of curves on the sphere. Consider the spherical embedding of the contracted graph $H$, and make the source vertices to be holes in the sphere. Two launch edges are topologically equivalent if and only if the curves of the edges are homeomorphic in this space. The definition using trivial partitions only allows us to compare two edges for equivalence in an obvious log-space manner instead of depending on a continuous space.

## 3 Reachability in LMPDs

Before describing the specifics of the algorithm for reachability, we describe the general concept in terms of a game on the contracted graph $H$.

Consider a board game for a single player with an oracle. The board is an undirected planar multigraph with the same vertex set and curves for edges except has exactly one representative edge for each topological class from $H$. Designate the vertex $u$ as the "start" and $v$ as the "finish". The game piece is a coin with an arrow painted on it to place on the vertices. Initially, the piece is placed on the start vertex.

The first move chooses an edge leaving the start vertex and the piece is moved to the other end of that edge, with the arrow on the coin pointing to the edge. The arrow will always point to an edge, called the *current edge*. For the remainder of the game, it is impossible to return to the start vertex unless the game is reset. All edges to the start vertex will be ignored.

There are two choices per move, Left or Right, and Stay or Cross.

The Left move turns the coin counterclockwise until the arrow points at a new edge. The Right move turns the coin clockwise until the arrow points at a new edge.

The Cross move attempts to move the coin across the current edge to the opposite vertex and point the arrow back at the current edge. However, the oracle will either accept or reject this move, based on the previous moves since leaving the start vertex. The Stay move does not attempt to move the coin, and the oracle is not used.

The game ends successfully when either the piece lands on the end vertex, or it is determined correctly that all attempts will fail.

How can one determine that all attempts will fail when the oracle's response at each Cross move depends on the previous steps? There is an added assumption that if a successful solution

exists, there exists a solution that takes at most $6m$ moves. Thus, all sequences of moves can be enumerated in finite time. Algorithm 3.1 defines the algorithm for solving the Coin Crawl game.

**Algorithm 3.1** The Coin Crawl Game

```
for all strings σ ∈ ({LEFT, RIGHT} × {CROSS, STAY})^{6m} do
  for all Start edges e_s do
    Move coin from the start to the other side of e_s, and point the arrow to e_s.
    for i = 1, ..., 6m do
      Rotate coin in direction σ_{i,1} from current edge.
      if σ_{i,2} = CROSS and Oracle allows a cross on the new edge then
        Move coin across edge.
      else
        Try the next starting edge e_s.
```

This method serves as a conceptual backdrop for the algorithm that solves reachability in planar DAGs with $m$ sources using $O(m + \log n)$ space. We now translate the actions in the coin crawl game to an algorithm our $m$-source graph $G$. First, we define the data structure representing the coin in Section 3.1 in terms of an explored region. This includes a hidden step of expanding an explored region around the current equivalence class. Second, Section 3.2 details how the limit on the number of moves required holds. Next, Section 3.3 defines how the Left, Right, Stay, and Cross moves modify the explored region, including testing if a move is possible, giving the oracle's response. Finally, Section 3.4 combines these methods into the final algorithm and solidifies missing details.

## 3.1 Exploring an Equivalence Class

Our first extension of SMPDExplore is to explore beyond a single tree and use edges of a single topological class in addition to tree, local, and jump edges. The method ExploreClass($e$) detailed in Algorithm 3.2 enumerates all vertices reachable from the endpoints of $e$ using tree, local, and jump edges and edges of the class $[e]$.

**Algorithm 3.2** ExploreClass

**Require:** Input edge $e = (x_1, x_2)$ within explored region $[\mathsf{LimLeft}_1, \mathsf{LimRight}_1]$.
**Ensure:** Enumerates all vertices reachable by edges in the explored region of the same class as $e$.

```
LimLeft_2 = FarthestLeft(x_2)
LimRight_2 = FarthestRight(x_2)
Updated = TRUE
while Updated do
  Updated = FALSE
  for all edges e' = (i, j) with i within LimLeft_1, LimRight_1 sharing class with e do
    SMPDExplore(j)
    if IsClockwise(LimLeft_1, i, x) and IsClockwise(LimRight_2, FarthestLeft(j), LimLeft_2) then
      LimLeft_2 = FarthestLeft(j)
      Updated = TRUE
    if IsClockwise(x, i, LimRight_1) and IsClockwise(LimRight_2, FarthestRight(j), LimLeft_2) then
      LimRight_2 = FarthestRight(j)
      Updated = TRUE
  for all edges e' = (i, j) with i within LimLeft_2, LimRight_2 sharing class with e do
    SMPDExplore(j)
    if IsClockwise(LimLeft_2, i, y) and IsClockwise(LimRight_1, FarthestLeft(j), LimLeft_1) then
      LimLeft_1 = FarthestLeft(j)
      Updated = TRUE
    if IsClockwise(y, i, LimRight_2) and IsClockwise(LimRight_1, FarthestRight(j), LimLeft_1) then
      LimRight_1 = FarthestRight(j)
      Updated = TRUE
```

We maintain an explored region by two pairs of variables $\mathsf{LimLeft}_a$ and $\mathsf{LimRight}_a$ for $a = 1, 2$. $\mathsf{LimLeft}_1$ and $\mathsf{LimRight}_1$ are the left- and right-most boundaries of the explored region surrounding the tail of $e$. Similarly, $\mathsf{LimLeft}_2$ and $\mathsf{LimRight}_2$ are the boundaries surrounding the head of $e$. It is important to note that if $e$ is a loop edge, these boundaries are on the same tree while launch edges have explored regions on different trees. Let $e = (x_1, x_2)$. If we do not already have an explored region to use, initialize $\mathsf{LimLeft}_1$ as $\mathrm{FarthestLeft}(x_1)$, $\mathsf{LimRight}_1$ as $\mathrm{FarthestRight}(x_1)$.

Each iteration attempts to expand the explored region by first taking all edges in class $[e]$ that have tails within $\mathsf{LimLeft}_1$ and $\mathsf{LimRight}_2$ and attempting to expand $\mathsf{LimLeft}_2$ and $\mathsf{LimRight}_2$ based on the SMPDExplore of their heads. Then, the expansion is reversed. This proceeds until both steps do not advance the explored region at all.

It is clear that each vertex in these trees reachable with these types of edges will be enclosed within the explored region by following the path it takes and knowing that all tree, local, and jump edges are enumerated and included in the explored region and the edges of class $[e]$ will expand the explored region as they are found. However, we need to ensure that no extra vertices are included, which is simple to see holds when the edges used to expand the explored region are reachable. We cannot guarantee that all edges in the explored region of class $[e]$ are reachable from $e$, but we can guarantee that the heads of those edges are reachable.

**Lemma 3.1.** *During any iteration of* ExploreClass*, all edges of class* $[e]$ *within the explored region have heads that are reachable from* $e$*.*

*Proof.* Before the expansion step of ExploreClass, all edges of class $[e]$ that are within the explored region are actually within SMPDExplore of the endpoints of $e$, so are reachable by Proposition 2.2. Proceed by induction on $i$ to show that the heads of all the edges of class $[e]$ within the explored region at iteration $i$ are reachable from $e$.

Let $e' = (a, b)$ be the an edge of class $[e]$ so that the $i$th iteration of ExploreClass has $e'$ in the explored region, but the $i-1$th iteration does not. This implies that there was an edge $f = (c, d)$ that had a tail within the $i-1$th explored region, but a head outside. Since $d$ is reachable from the induction hypothesis, so is SMPDExplore($d$). If $a$ or $b$ is in SMPDExplore($d$), then $b$ is reachable from $e$. Assume not.

Since $e'$ has one endpoint in the explored region, from the viewpoint of at least one source tree, the edges $e, e'$, and $f$ appear in clockwise order, since their endpoints will appear in clockwise order. This requires that the path from $e$ to $f$ following tree, local, jump, and class $[e]$ edges must cross the tree path from one of the sources to one of the endpoints of $e'$. Hence, at least one endpoint of $e'$ is a descendant of a vertex in this path, and then the head $b$ is reachable in either case. □

The remaining necessary property is stated as the following proposition.

**Proposition 3.2.** ExploreClass($e$) *results in an explored region so that any launch edge $f$ within the explored region and of a different class than e is reachable from e.*

*Proof.* Let $f$ be an edge of a different class within the explored region. Although all cases have an identical argument, consider $f$ to be within $\mathsf{LimLeft}_1$ and $\mathsf{LimRight}_1$, and clockwise of $e$. Then, if $e = (x_1, y_1)$ and $f = (x_2, y_2)$, IsClockwise$(x_1, x_2, \mathsf{LimRight}_1) =$ TRUE. By Lemma 2.4, there are no edges of class $[e]$ with vertices clockwise of $f$ and counterclockwise of $\mathsf{LimRight}_2$. Hence, $f$ is in the SMPDExplore of the last edge to expand $\mathsf{LimRight}_1$. Since the head of that edge is reachable from $e$, so is $f$. □

### 3.2 $m$ sources require at most $O(m)$ moves

Now that we have a method for searching within a topological equivalence class, we demonstrate that we will not need to search more than a linear number of classes, with respect to the number of sources. The statement is simply this: if a path from $u$ to $v$ exists, there exists a path that uses each topological class at most twice. This means the path can be partitioned so that each part contains only tree, local, jump, and edges of the same topological class, and each topological class appears in at most two of these parts.

**Definition 3.3.** Given a path $P$ from $u$ to $v$ in $G$, describe $P$ by the edges $e_1, \ldots, e_k$. A set of $\ell$ integers $i_1 < \cdots < i_\ell = k$ partition $P$ into sub-paths $P_j = e_{i_j}, \ldots, e_{i_{j+1}-1}$ for $j = 1, \ldots, k-1$. $P$ has an *$\ell$-order topological partition* if there are $\ell$ such integers so that each of these sub-paths $P_j$ consist of edges that are either tree, local, and jump edges, or are all of the same topological class.

There is a clean description of a property for paths that allows us a lot of control over its behavior in the contracted graph.

**Definition 3.4.** Let a path $P$ follow the vertex sequence $x_1, \ldots, x_\ell$. $P$ is an *irreducible path* if every pair $x_i, x_j$ where $i < j$ and $x_j$ is a descendant of $x_i$ in a source tree, then $P$ follows the tree path from $x_i$ to $x_j$.

It is clear that if a $uv$ path exists, then an irreducible $uv$ path exists by deleting subpaths between pairs $x_i, x_j$ that violate the definition, and inserting the tree path.

**Lemma 3.5.** *If there exists a path from $u$ to $v$, there exists a path $P$ from $u$ to $v$ with an $\ell$-order topological partition for some $\ell \leq 6m$.*

*Proof.* Assume there exists a path from $u$ to $v$, but there does not exist one that fits the theorem. Let $P = e_1 \ldots e_k$ be an irreducible $uv$ path with a minimum-order topological partition $i_1 < \cdots < i_\ell$. Since $\ell > 6m$, but there are less than $3m$ topological classes, there is some topological class $C$ repeated at least three times in blocks of $P$, we will show that this contradicts the irreducibile property.

We may assume that $P$ does not revisit $T_u$, since there is a tree-path from $u$ to the last vertex in both $P$ and $T_u$, we can substitute this new path.

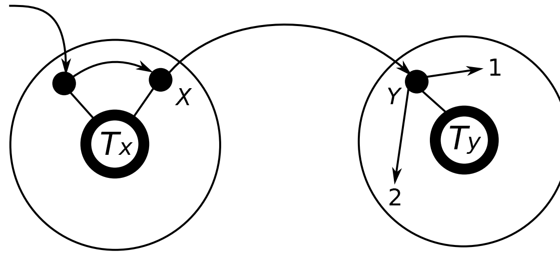



Figure 2: The choice of direction.

Now, consider $e_{j_1} = (x_{j_1}, y_{j_1})$, the first edge of class $C$ that appears in $P$. Let $T_x$ be the source tree containing $x_{j_1}$ and $T_y$ the source tree containing $y_{j_1}$. Define $S_x$ and $S_y$ as the sources for those trees. The path $P$ visits $T_x$ first by some launch or loop edge of a different class, since $P$ starts in $T_u$ and never returns to that tree. Without loss of generality, assume $P$ travels clockwise from this edge to $e_{j_1}$. Otherwise, mirror the plane and the argument is identical. The edge $e_{j_1}$ and the tree path from $S_y$ to the edge gives a boundary forcing the path either clockwise or counterclockwise in the tree $T_y$. We split cases depending on this direction.

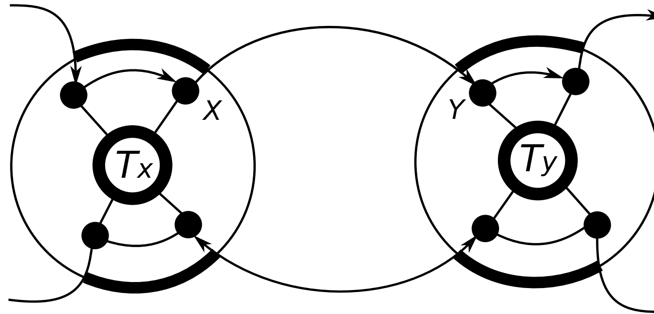



Figure 3: Case 1.

Case 1: The edge following $e_{j_1}$ is in the clockwise direction. Since the path after $e_{j_1}$ cannot enter this area and cannot cross the tree paths without creating a cycle, the next launch or loop

edge must be of a different class. When the next edge $e_{j_2}$ with class $C$ appears, it does not matter which direction it travels, but will give a closed region with $e_{j_1}$ where the path cannot enter without creating a cycle or violating $(*)$. Recall Lemma 2.4 that gives all edges of class $C$ appearing contiguously around $T_x$ and $T_y$ when listing the launch and loop edges in clockwise order. Thus the third appearance of the class $C$ must be between the edges $e_{j_1}, e_{j_2}$ and the edges of other classes. See Figure 3, where thick lines represent regions of the trees where all reachable vertices are descendants of the path, which cover all possible choices for this third class, violating the irreducible property.

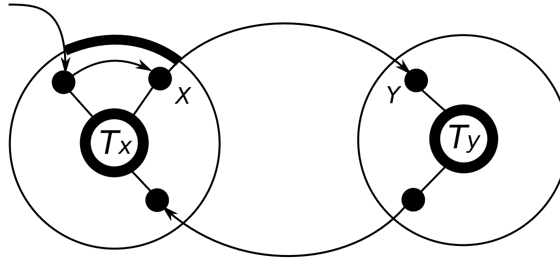



Figure 4: Case 2.

Case 2: The edge following $e_{j_1}$ is in the counterclockwise direction. This develops two sub-cases by the direction of the last edge $e'$ of class $C$ in this block. Note that the path cannot reenter the closed region given by the edges $e_{j_1}$ and $e'$ without creating a cycle. Moreover, the path would not be able to leave the region without violating the irreducible property.

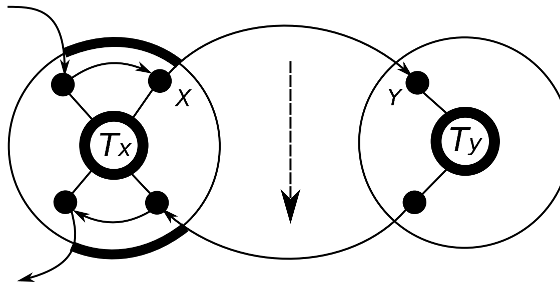



Figure 5: Case 2.A.

Case 2.A: $e'$ is from $T_y$ to $T_x$. See by the thick lines in Figure 5 that any second appearance of an edge in class $C$ must be a descendant of the path in $T_x$, violating the irreducible property.

Case 2.B: $e'$ is from $T_x$ to $T_y$. See by the thick lines in Figure 6 that any second appearance of class $C$ must either be a descendant of the path on $T_x$ or the path on $T_y$, violating the irreducible property.

Since all cases contradict our assumption, it is not possible for a class to appear in more than two separate blocks. □

Note that now, we have a constant upper bound on the number of times a topological class can be repeated, which allows our search to visit each class at most twice. Combine this with the linear

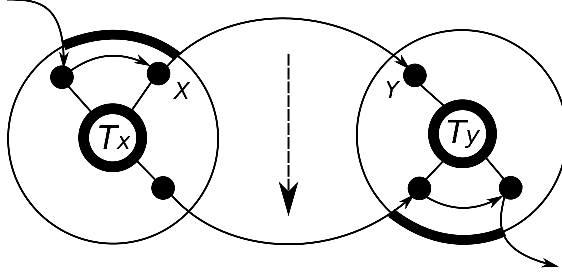



Figure 6: Case 2.B.

bound on the number of possible classes in a planar graph with respect to the number of sources, and if a $uv$ path exists, there is a path that has a linear-order topological partition. This relates to the upper bound on moves required in the Coin Crawler game, as there are at most $3m - 6$ topological classes of launch or loop edges, and each is required at most twice, leaving at most $6m$ times we will need to explore a topological class, corresponding to Go moves.

**Lemma 3.6.** *If there exists a path from u to v in G, there exists a starting edge $e_s$ and a sequence of moves $\sigma$ of length at most* $6m$ *enumerated by* Reach *that leads the algorithm from to v.*

*Proof.* By Lemma 3.5, if there exists a $uv$ path, there exists some irreducible path $P$ with a $6m$-order topological partition. By this assumption, we will have a single edge leaving $T_u$ and will never return to this tree.

Now, consider the list of launch and loop edges in $P$. Label these edges as $e_s, e_1, \ldots, e_k$. Note that the first launch edge must leave $T_u$, and thus $e_s$ is a proper choice of labeling. Now, let $e'_{i,1}$ and $e'_{i,2}$ denote the first and last edges of the $i$th class block in the topological partition of $P$. Note that $e'_{1,1} = e'_{1,2} = e_s$ for the first class. Assume we have a string $\sigma$ of length $k_i$ that brings the Coin Crawl gam We will produce a sequence of moves to append to $\mathbf{a}$ that will direct Reach to the edge $e'_{i+1,1}$. By repeated iteration, these moves will direct Reach between edges $e'_{i,2}$ to $e'_{i+1,1}$, while the ExploreClass method directs the edge $e'_{i,1}$ to $e'_{i,2}$, as the only edge in $P$ between these edges are tree, local, jump edge or share the same topological class.

Given an edge $e'_{i,2}$, we must have that the edge $e'_{i+1,1}$ is within SMPDExplore of the head vertex in $e'_{i,2}$ by the definition of the partition. Thus, if we construct a sequence of moves that direct Reach to the same class as $e'_{i+1,1}$, the ExploreClass method will see $e'_{i+1,1}$ within the explored region, and enumerate it. The path $P$ has a subpath $P_{i,i+1}$ from $e'_{i,2}$ to $e'_{i+1,1}$ consisting of tree, local, and jump edges within a source tree $T_j$. This subpath follows either a clockwise or counterclockwise path around $T_j$.

There is a sequence of launch and loop classes that are descendants of $P_{i,i+1}$. By Lemma 2.4, these classes appear in blocks, one block per launch class and at most two blocks for loop classes. Let $s_i$ be the number of such blocks, not including those edges of the classes $e'_{i,2}$ and $e'_{i+1,1}$. If the direction of $P_{i,i+1}$ is clockwise, append $s_i$ copies of the (Right,Stay) move. Otherwise, append $s_i$ copies of the (Left,Stay) move. We will append another move of the same rotational direction in order to reach $e'_{i+1,1}$, but the Cross or Stay choice is determined by $e'_{i+1,2}$.

The edges of class $e'_{i+1,1}$ appear in two blocks, and the edges span between these blocks. The Reach algorithm considers the block with the tail of $e'_{i+1,1}$ to be intersecting the explored region defined by $\mathsf{LimLeft}_1$ and $\mathsf{LimRight}_1$. If $e'_{i+1,2}$ has head in the other block, we require a Cross move

to change the first explored region, which corresponds to the location of the "coin," to the other block. Otherwise, we will have a STAY move.

This process constructs a list of moves $\sigma$ that will allow Reach to "crawl" across $G$ in the directions given by $P$ and finally reach an edge to $v$. If the length of $\sigma$ is less than $6m$ we can add extra moves to the end to make it have the proper length given by the algorithm. The final step of this proof is to show that the length of $\sigma$ is actually of length less than $6m$.

Each move in $\sigma$ is classified into two types: a move that is incoming to an edge class in $P$, and a move that skips an edge class, as we rotate around a source tree. When a move skips a class, this implies that the subpath $P_{i,i+1}$ corresponding to this move is an ancestor of *all* edges in that class. Thus, either there is a cycle in $G$, or $P$ has a later vertex that is a descendant of these vertices, but does not follow the tree path. Thus, $P$ cannot contain edges in this class.

Let $\ell$ be the number of classes appearing in $P$. Each of these classes contribute a single incoming move to $\sigma$ for each partition they contribute to $P$. By choice of $P$, this is at most $2\ell$ incoming moves. Moreover, there are at most two skip moves for each class, giving $2k$ skip moves, if there are $k$ edge classes not appearing in $P$. Since $\ell + k$ is at most $3m - 6$, by Theorem 2.6, we have at most $6m$ moves in $\sigma$. □

### 3.3 Navigating Adjacent Classes

It has been shown that we can compare two edges and determine if they have the same topological class in logarithmic space. However, in order to enumerate the topological classes, we must store a representative edge of each previously encountered class in order to prevent duplicate counts. This gives us $O(m \log n)$ space for graphs with $m$ sources.

In the previous section, it was shown that if a $u-v$ path exists, there exists one with a $m$-order topological partition. Our goal is to determine enough information to cover all possible $m$-order partitions within $O(m)$ space. However, explicitly listing the classes for each block in a partition would result in $O(m \log n)$ space. Not all classes can be reached consecutively, so instead we only care about adjacent classes. That is, classes that reach each other within the source trees they share.

Note Lemma 2.4 guarantees all launch edges of the same class appear in consecutive order when the launch and loop edges in a source tree are listed in clockwise order. This defines boundaries between blocks of launch and loop classes and the notion of a *clockwise (counterclockwise) neighbor* of the class $C$ can be defined as the class immediately following (preceding) the edges of class $C$ in the clockwise ordering. It is important to start from a specific endpoint of an edge of the class $C$, as there are either two trees of class $C$ in the case of launch edges or two blocks in the clockwise ordering in the case of loop edges.

We define the method NextClass as Algorithm 3.3 to return a representative edge of the class adjacent to a given class in a certain direction from a given edge and endpoint. The algorithm inputs a launch or loop edge $e$ with endpoint $x$ and direction LEFT or RIGHT. It iterates over all launch and loop edges on the source tree of $x$ and tracks the closest edge to the $x$ that is not in the class $[e]$. The closest edge depends on the direction, using the IsClockwise relation properly. This algorithm performs LEFT and RIGHT moves in the Coin Crawl game.

**Algorithm 3.3** NextClass($e_0, x, \mathsf{Dir}$)

**Require:** Input edge $e_0$ with endpoint $x$ and direction $\mathsf{Dir}$.

**Ensure:** Returns an edge $e'$ representing the class adjacent to $e_0$ to the direction given by $\mathsf{Dir}$ in the clockwise ordering around $x$.

```
e' = NULL, z = NULL
for all Launch or loop edges e = (a_1, a_2) do
  for i = 1, 2 do
    if a_i shares the source tree with x then
      if Dir = RIGHT then
        if z = NULL or IsClockwise(x, a_i, z) then
          if e and e_0 are of the same class then
            if IsClockwise(x, a_{i+1}, a_i) then
              z = a_i, e' = e
          else
            z = a_i, e' = e
      else
        if z = NULL or IsClockwise(z, a_i, x) then
          if e and e_0 are of the same class then
            if IsClockwise(a_i, b, x) then
              z = a_i, e' = e
          else
            z = a_i, e' = e
Return e'.
```

### 3.4 Reach for LMPD in $L$

Now, we have the pieces in place to translate the Coin Crawl game into a complete algorithm. Algorithm 3.4 combines the local operations ExploreClass and NextClass into an exhaustive global search over all possible move choices.

**Theorem 3.7.** *Reachability for $m$-source Multiple-sink Planar DAGs can be computed in $O(m + \log n)$ space using Algorithm 3.4.*

*Proof.* Let the input graph $G$ be as described previously, and find the spanning forest similarly. Note that $\sigma \in (\{\text{Left, Right}\} \times \{\text{Cross, Stay}\})^{6m}$, denotes a move as $\sigma_i \in \{\text{Left, Right}\} \times \{\text{Cross, Stay}\}$, so $\sigma_{i,1}$ is a direction and $\sigma_{i,2}$ designates a cross or stay move. We maintain two explored regions, one on each end of the current topological class. Follow this sequence of moves and use ExploreClass at every topological class the coin points to, using NextClass to navigate adjacent classes. If there is an edge to $v$ inside the explored region at any point, return successfully, Proposition 3.2 guarantees a path that follows these moves to that edge, so successful completion implies reachability. In order to ensure successful completion when a path exists, the following claim is necessary.

**Claim 3.1.** *If there exists a path from $u$ to $v$ in $G$, there exists a starting edge $e_s$ and a sequence of moves $\sigma$ of length at most* $6m$ *that leads* Reach *from $u$ to $v$.*

**Algorithm 3.4** Reach$(G, u, v)$

**Require:** Input connected planar DAG $G$ with at most $m$ sources, vertices $u, v \in V(G)$.
**Ensure:** Returns TRUE if and only if there is a path from $u$ to $v$ in $G$.

```
for all strings σ ∈ ({LEFT, RIGHT} × {CROSS, STAY})^{6m} do
  for all edges e_s = (x, y) leaving T_u do
    LimLeft_1 = LimLeft_2 = FarthestLeft(y)
    LimRight_1 = LimRight_2 = FarthestRight(y)
    Center_1 = Center_2 = y
    Current = e_s
    for all i = 1, ..., 6m do
      if there is an edge e' from the explored region to v then
        There is a path from u to v
        Return TRUE
      else
        Attempt to move in the direction σ_{i,1}.
        Current = NextClass(Current, Center_1, σ_{i,1})
        Expand the explored region along the new edge class
        ExploreClass(Current)
        if σ_{i,2} = CROSS then
          if ∃e' = (a, b) of class Current with IsClockwise(LimLeft_1, a, LimRight_1) = TRUE then
            Center_1 = a
            Center_2 = b
            Swap(LimLeft_1, LimLeft_2)
            Swap(LimRight_1, LimRight_2)
            Swap(Center_1, Center_2)
          else
            Try the next starting edge e_s.
```

*Proof of Claim.* By Lemma 3.5, if there exists a $uv$ path, there exists some irreducible path $P$ with a $6m$-order topological partition. By this assumption, we will have a single edge leaving $T_u$ and will never return to this tree.

Now, consider the list of launch and loop edges in $P$. Label these edges as $e_s, e_1, \ldots, e_k$. Note that the first launch edge must leave $T_u$, and thus $e_s$ is a proper choice of labeling. Now, let $e'_{i,1}$ and $e'_{i,2}$ denote the first and last edges of the $i$th class block in the topological partition of $P$. Note that $e'_{1,1} = e'_{1,2} = e_s$ for the first class. Assume we have a string $\sigma$ of length $k_i$ that brings the Reach algorithm to the edge $e'_{i,1}$. We will produce a sequence of moves to append to $\sigma$ that will direct Reach to the edge $e'_{i+1,1}$. By repeated iteration, these moves will direct Reach between edges $e'_{i,2}$ to $e'_{i+1,1}$, while the ExploreClass method expands the explored region to include $e'_{i+1,1}$, as the only edge in $P$ between $e'_{i,1}$ and $e'_{i+1,1}$ are tree, local, jump edge or have topological class $[e'_{i,1}]$.

Given an edge $e'_{i,2}$, we must have that the edge $e'_{i+1,1}$ is within SMPDExplore of the head vertex in $e'_{i,2}$ by the definition of the partition. The path $P$ has a subpath $P_{i,i+1}$ from $e'_{i,2}$ to $e'_{i+1,1}$ consisting of tree, local, and jump edges within a source tree $T_j$. This subpath follows either a clockwise or counterclockwise path around $T_j$.

There is a sequence of launch and loop classes that are descendants of $P_{i,i+1}$. By Lemma 2.4, these classes appear in blocks, one block per launch class and at most two blocks for loop classes. Let $s_i$ be the number of such blocks, not including those edges of the classes $e'_{i,2}$ and $e'_{i+1,1}$. If the direction of $P_{i,i+1}$ is clockwise, append $s_i$ copies of the (RIGHT,STAY) move. Otherwise, append $s_i$ copies of the (LEFT,STAY) move. These $s_i$ moves will be referred to as *skip* moves, as they do not require edges of those classes in order to rotate farther in the subtree to the next topological class. We will append another move of the same rotational direction in order to reach $e'_{i+1,1}$, but the CROSS or STAY choice is determined by $e'_{i+1,2}$.

If the tail of $e'_{i+1,2}$ is on the same source tree as the tail of $e'_{i+1,1}$, we require a CROSS move to change the first explored region, which corresponds to the location of the coin, to the other block. Otherwise, we will have a STAY move.

When a move skips a class between edges $e'_{i,2}$ and $e'_{i+1,1}$, this implies that the subpath $P_{i,i+1}$ corresponding to this move is an ancestor of *all* edges in that class. Thus, either there is a cycle in $G$, or $P$ has a later vertex that is a descendant of these vertices, but does not follow the tree path, violating the irreducible property. Thus, $P$ cannot contain edges in this class.

Let $\ell$ be the number of classes appearing in $P$. Each of these classes contribute a single incoming move to $\sigma$ for each partition they contribute to $P$. By choice of $P$, this is at most $2\ell$ incoming moves. Moreover, there are at most two skip moves for each class, giving $2k$ skip moves, if there are $k$ edge classes not appearing in $P$. Since $\ell + k$ is at most $3(m + 2) - 6$, by Theorem 2.6, we have at most $6m$ moves in $\sigma$. □

Thus, when a path exists, Reach will return enumerate the move sequence generated in this claim and return with success. □

Finally, setting $m = O(\log n)$, Theorem 1.1 follows.

# 4 Recognition of LMPDs

It remains to show that we can detect if a graph is a log-source multiple-sink planar DAG. We have the ability to count the number of sources and find a planar embedding. Allender et al. can be used to check that each graph induced by a source tree $T_i$ is acyclic. Now, we need to see if there is a cycle that spans multiple trees.

Suppose $G$ has $m$ sources with the choice of forest so that each induced subgraph for the source trees is acyclic, but there exists a cycle that spans multiple trees. Then, there exist vertices $x$ and $y$ from different trees that lie in a cycle that uses the fewest number of loop and launch edges. This cycle can be reduced to an *irreducible cycle*, a similar definition to an irreducible path: if the launch edge is from $x$ to $y$, then there is a path from $y$ to $x$ along the cycle. This path can be made irreducible, which creates an irreducible cycle.

For an irreducible cycle, it is impossible for the path from $y$ to $x$ to pass through the tree paths from previous vertices to their sources, and the proof of Lemma 3.5 holds. Thus, the algorithm for reachability in LMPDs will find this path from $y$ to $x$.

Note, this method does not definitively check if there is a cycle between two vertices, but will return true at least in the case that the vertices are part of the minimal-length cycle.

In order to check for cycles in a graph $G$ with $m$ sources, we can iterate over all launch edges $e = (x, y)$ and check for $yx$-reachability in $G - e$ using $O(m + \log n)$ space. If no cycles are found, we can be sure the graph is acyclic.

## References


[1] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Grid graph reachability problems. In *21nd Annual IEEE Conference on Computational Complexity*, pages 299–313, 2006.

[2] Eric Allender and Meena Mahajan. The complexity of planarity testing. *Information and Computation*, 189:117–134, 2004.

[3] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in $\mathsf{NC}^1$. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[4] David A. Mix Barrington, Chi-Jen Lu, Peter Bro Miltersen, and Sven Skyum. Searching constant width mazes captures the $\mathsf{AC}^0$ hierarchy. In *15th International Symposium on Theoretical Aspects of Computer Science (STACS)*, Volume 1373 in Lecture Notes in Computer Science, pages 74–83. Springer, 1998.

[5] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. In *22nd Annual IEEE Conference on Computational Complexity*, pages 217–221, 2007.

[6] Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. A log-space algorithm for canonization of planar graphs. *CoRR*, abs/0809.2319, 2008.

[7] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400–411, June 1997.

[8] Andreas Jakoby, Maciej Liśkiewicz, and Rüdiger Reischuk. Space efficient algorithms for directed series-parallel graphs. *J. Algorithms*, 60(2):85–114, 2006.

[9] Andreas Jakoby and Till Tantau. Logspace algorithms for computing shortest and longest paths in series-parallel graphs. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 2007.

[10] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[11] Omer Reingold. Undirected st-connectivity in log-space. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 376–385. IEEE Computer Society Press, 2005.

[12] Douglas B. West. *Introduction to Graph Theory.* Prentice-Hall, second edition, 2001.