Simultaneous Space-Time Complexities of Variants of Reachability

A B.Tech Project Report

Submitted by

Vaibhav Agarwal (B21CS077)
Alok Kumar (B21CS006)

Under the Supervision
of
Dr. Vimal Raj Sharma

Department of Computer Science Engineering

Indian Institute of Technology Jodhpur

December, 2024-2025

Details

Project Title :	Simultaneous Space-Time Complexities of Variants of Reachability
Submitted by :	Vaibhav Agarwal (B21CS077) Alok Kumar (B21CS006)
Supervisor :	Dr. Vimal Raj Sharma
Institution :	Indian Institute of Technology Jodhpur
Department :	Computer Science and Engineering
Degree Program :	B.Tech in Computer Science
Month of Submission :	December 2024
Project Objective :	To write a survey for the following three research papers:

(i) An $O(n^{1/2+\epsilon})$-Space and Polynomial-Time Algorithm for Directed Planar Reachability

(ii) A Sublinear Space, Polynomial Time Algorithm for Directed s-t Connectivity

(iii) Simultaneous Time-Space Upper Bounds for Red-Blue Path Problem in Planar DAGs

Acknowledgement

We would like to extend our heartfelt gratitude to our supervisor, **Dr. Vimal Raj Sharma**, whose unwavering support, guidance, and mentorship have been pivotal in the successful completion of this report. Their profound expertise and deep understanding of graph theory, particularly in the area of graph reachability, have greatly enriched our perspective and inspired us to explore this fascinating field with curiosity and rigor.

Throughout this journey, **Dr. Vimal** has consistently provided invaluable feedback, constructive criticism, and encouragement, which have not only enhanced the quality of this report but also deepened our understanding of the research papers and their implications. Their insightful advice has guided us through challenges, and their ability to simplify complex concepts has been instrumental in helping us grasp intricate details and build a coherent analysis.

Beyond their technical expertise, we are grateful for their patience, encouragement, and willingness to invest their time and energy in mentoring us. Their belief in our abilities and dedication to fostering a collaborative and supportive learning environment have motivated us to strive for excellence and take pride in our work.

This report would not have been possible without their guidance, and we are sincerely thankful for their constant support throughout the process. It has been a privilege to learn under their mentorship, and their influence will continue to inspire our academic and professional pursuits.

Preface

Graph reachability, particularly in directed graphs, plays a pivotal role in theoretical computer science and practical applications such as routing, database queries, and verification systems. This report examines three foundational papers that propose innovative approaches to solving reachability problems under specific constraints, such as sublinear space, polynomial time, and planar graph structures.

The selected papers are:

1. **An $O(n^{1/2+\epsilon})$-Space and Polynomial-Time Algorithm for Directed Planar Reachability -** This paper presents a polynomial-time reachability algorithm for directed planar graphs that achieves $O(n^{1/2+\epsilon})$- space bound.

2. **A Sublinear Space, Polynomial Time Algorithm for Directed *s-t* Connectivity -** This paper focuses on designing an algorithm for directed *s-t* connectivity that uses sublinear space while maintaining polynomial time complexity.

3. **Simultaneous Time-Space Upper Bounds for Red-Blue Path Problem in Planar DAGs -** This paper exhibit a polynomial time and $O(n^{1/2+\epsilon})$ space algorithm for the RedBluePath problem and EvenPath problem in planar DAG.

An $O(n^{1/2+\epsilon})$-Space and Polynomial-Time Algorithm for Directed Planar Reachability

ABSTRACT

This paper shows that the reachability problem over directed planar graphs can be solved simultaneously in polynomial time and approximately ***O(√n)*** *space. In contrast, the best space bound known for the reachability problem on general directed graphs with polynomial running time is* ***O(n/2*** $^{\sqrt{\log n}}$ ***)****.*

INTRODUCTION

The graph reachability problem is to decide, for a given graph G and its two vertices s and t, whether there is a path from s to t in G. This problem is central to space bounded computations.

The standard breadth first search algorithm (BFS) and Savitch's algorithm are two of the most fundamental algorithms known for solving directed graph reachability. BFS can be implemented in linear time and space. Savitch's algorithm only takes $O(\log^2 n)$ space but requires $\Theta(n^{\log n})$ time. BFS is efficient in time but not in space and Savitch's algorithm is efficient in space but takes super-polynomial time.
Hence a natural and significant question that arises is whether we can design an algorithm for the directed graph reachability problem that is efficient in both time and space. In particular, can we design a polynomial-time algorithm for the directed graph reachability problem that uses only $O(n^{1-\epsilon})$ space for some small constant ϵ ?

This paper presents a polynomial-time reachability algorithm for directed planar graphs that achieves $O(n^{1/2+\epsilon})$*- space bound.*

Theorem 1. *For any constant* $0 < \epsilon < 1/2$*, there is an algorithm that, given a directed planar graph G and two vertices s and t, decides whether there is a path from s to t. This algorithm runs in time* $n^{O(1/\epsilon)}$ *and uses* $O(n^{1/2+\epsilon})$ *space, where n is the number of vertices of G.*

For proving the above theorem we first give a polynomial- time and $\tilde{O}(\sqrt{n})$-space algorithm for computing a "separator" of $O(\sqrt{n})$ size for an undirected planar graph ($\tilde{O}(s(n))$ means $O(s(n)(\log n)^{O(1)})$).
This result can be stated as a theorem below.

Theorem 2. *There is an algorithm that takes an undirected planar graph G with n vertices as input and outputs a (8/9)- separator of G. This algorithm runs in polynomial time and uses* $\tilde{O}(\sqrt{n})$ *space.*

We call the above algorithm Separator. Our reachability algorithm uses algorithm Separator. It is important to note that though our reachability algorithm works on a directed planar graph, for constructing the separator we work on the underlying undirected graph. For a given directed planar graph G, the reachability algorithm first applies Separator to its underlying undirected graph and computes a small size "separator family" S — a set of vertices, removal of which splits G into several disconnected subgraphs of smaller size. The crucial observation is that any path in G from one component to another has to go through S. Based on this, we construct a new directed graph H on S. There is a directed edge (a, b) in H if there is a path from a to b that goes through a single component. This can be decided by solving reachability on a smaller planar graph. This implies that reachability in G reduces to reachability in H. Since H is a smaller graph we can afford to perform BFS on H. This leads to a polynomial-time, $O(n^{2/3})$-space algorithm. To obtain the required $n^{1/2+\epsilon}$-space bound we apply this idea recursively. By ensuring that the depth of recursion is a constant, we can ensure that the running time of the algorithm remains polynomial.

PRELIMINARIES

We will use the standard notions and notation for algorithms, complexity measures, and graphs. For any set X, |X| denotes the number of elements in X. By log we mean the base 2 logarithm.

If G = (V, E) is directed graph, by $\underline{G}$ we mean its underlying undirected graph. We use 'n' to denote the number of vertices of an input graph G. For any $U \subseteq V$, G(U) denote the subgraph of G induced by U.
The notion of separator is central throughout the paper. So, we define this notion formally.

Definition 1. *For any undirected graph G and for any constant ρ, $0 < \rho < 1$, a subset of vertices S is called a ρ-separator if (i) removal of S disconnects G into two subgraphs A and B, and (ii) the number of vertices of any component is at most ρn. The size of a separator is the number of vertices in the separator.*

REACHABILITY ALGORITHM GIVEN A SEPARATOR

First we define an algorithm that uses Separator iteratively to compute a separator family that splits a given graph into sublinear size components.

Definition 2. *For any undirected graph G with n vertices, an r(n)-separator family of G is a set $\overline{S}$ of vertices of G so that the removal of $\overline{S}$ disconnects G into subgraphs each of which contains at most r(n) vertices.*

By using Separator, for any $\epsilon > 0$, we can design an algorithm that produces an $n^{1-\epsilon}$-separator family of $O(n^{1/2+\epsilon/2})$ size in polynomial time and $\tilde{O}(n^{1/2+\epsilon/2})$ space.

Lemma 3. *For any $\epsilon > 0$, there is an algorithm* **SepFamily** *that takes a planar graph as input and outputs an $n^{1-\epsilon}$ - separator family of size $O(n^{1/2+\epsilon/2})$ in polynomial time and $\tilde{O}(n^{1/2+\epsilon/2})$ space.*

Proof: The key tool is the separator algorithm **Separator** of Theorem 2. For a given undirected planar graph of size 'n', this algorithm produces a (8/9)-vertex separator of size at most $c\sqrt{n}$ (for some constant $c > 0$); that is, removal of vertices in the separator disconnects the graph into two subgraphs of size $\leq 8n/9$. Intuitively, we repeat this procedure for some sufficient number of times so that each connected component contains at most $n^{1-\epsilon}$ vertices. The union of separators increases monotonically, and, as we will see, its size is bounded by $O(n^{1/2+\epsilon/2})$. This whole procedure can be implemented by using $\tilde{O}(n^{1/2+\epsilon/2})$ space for keeping the obtained separators.

For this, fix any input undirected graph $G = (V, E)$ with 'n' vertices. We first define an algorithm **Separator$^+$** so that it can be used iteratively. This algorithm takes G, a parameter *i* indicating the number of iterations, and a separator family $\overline{S}_i$ obtained so far, and it outputs a new separator family $\overline{S}_{i+1} := \overline{S}_i \cup \overline{S}'$. Here we may assume (by induction hypothesis) that each connected component of $G(V\backslash\overline{S}_i)$ is of size $\leq n_i := (8/9)^i n$. The set $\overline{S}'$ added by the algorithm is the union of vertex separators $S'_1, ..., S'_\ell$ of size $c\sqrt{n_i}$. Each S'_j is obtained by applying Separator to a connected component of $G(V\backslash\overline{S}_i)$ of size $> (8/9)n_i$. This guarantees the induction hypothesis on the size of connected components. More specifically, for every vertex $v \in V\backslash\overline{S}_i$, the algorithm first checks whether v is the vertex with the smallest index in the component G_v connected to v in $G(V\backslash\overline{S}_i)$, and if so and if the size of G_v is larger than $(8/9)^{i+1}n$, then **Separator** is applied to this component to produce a vertex separator S'_j for G_v.
Applying this algorithm **Separator$^+$** for k times, where k is defined by $k = \lceil (\epsilon \log n) / \log(9/8) \rceil$

$$k = \left\lceil \frac{\epsilon}{\log(9/8)} \log n \right\rceil$$

so that

$$\frac{1}{2} n^{1-\epsilon} \leq n \left(\frac{8}{9}\right)^k \leq n^{1-\epsilon}$$

holds. Thus, after the k applications of **Separator$^+$**, the size of connected components is at most $n^{1-\epsilon}$ as desired. Next we bound the size of the obtained separator family. Note that there are at most $(9/8)^{i+1}$ components of size $\geq (8/9)^{i+1}n$; hence, there are at most $(9/8)^{i+1}$ vertex separators added to $\overline{S}_i$ at the *i*th iteration, and each such vertex separator size is at most $c\sqrt{n_i} = c\sqrt{(8/9)^i n}$. Thus, the number of vertices in the final separator family $\overline{S}_{k+1}$ is bounded by

$$\left(\frac{9}{8}\right) c\sqrt{n} + \left(\frac{9}{8}\right)^2 c\sqrt{\left(\frac{8}{9}\right) n} + \cdots + \left(\frac{9}{8}\right)^{k+1} c\sqrt{\left(\frac{8}{9}\right)^k n}$$

$$= \frac{9c\sqrt{n}}{8} \sum_{i=0}^{k} \left(\frac{9}{8}\right)^{k/2} \leq c'\sqrt{n} \left(\frac{9}{8}\right)^{k/2} \leq 2c' n^{1/2+\epsilon/2}$$

for some constant c' > 0. Finally, observe that the space used is dominated by the space required to store the separator family. It is easy to see that the running time is bounded by a polynomial.

Now, consider any constant $\epsilon > 0$. We may assume that $\epsilon < 1/2$, because otherwise our target bounds, $\tilde{O}(n^{1/2+\epsilon/2})$ become trivial. Consider G = (V, E), s, and t be the given input; that is, G is a directed graph, and s and t are start and goal vertices in V . As outlined in the introduction, our algorithm first uses **SepFamily** to compute an $n^{1-\epsilon}$ -separator family $\overline{S}$ of size $O(n^{1/2+\epsilon/2})$ for the underlying undirected graph $\underline{G}$, and explores a path from s to t through vertices in $\overline{S}$. For any vertices a and b in $\overline{S}$, the reachability from a to b is determined by the reachability in $G(V' \cup \overline{S})$, where V' is the set of vertices of some connected component of $\underline{G}(V \setminus \overline{S})$ that is adjacent to both a and b in $\underline{G}(V' \cup \overline{S})$. (Thus, we can immediately determine that b is not reachable from a if there is no such connected component V' of $\underline{G}(V \setminus \overline{S})$.) For checking this connectivity, we may use the standard algorithm BFS for the reachability problem if $G(V' \cup \overline{S})$ is small enough. Although Lemma 3 guarantees that $|V' \cup \overline{S}| = O(n^{1-\epsilon})$, it is still large to execute BFS. Instead we use our algorithm recursively on $G(V' \cup \overline{S})$. Note further that |V'| is too large to store; thus, V' (and hence, $G(V' \cup \overline{S})$) must be given implicitly. In the algorithm, we specify V' as a set of vertices connected to some vertex in $\underline{G}(V \setminus \overline{S})$; then we can check whether $v \in V'$ for a given v by using the undirected graph reachability algorithm **UReach** in O(log n) space and polynomial time. Hence, the algorithm can be modified to handle such implicitly given graphs with the same order of space complexity.

Algorithm for Directed Planar Reachability

1. **PlanarReach**($\hat{G}$, $\hat{s}$, $\hat{t}$, n)
   (let $\hat{n}$ be the number of vertices of $\hat{G}$)
2. **If** $\hat{n} \leq n^{1/2}$
3. **then BFS**($\hat{G}$, $\hat{s}$, $\hat{t}$)
4. **Else** (let $\hat{r} = \hat{n}^{1-\epsilon}$)
5. Run **SepFamily** to compute $\hat{r}$-separator family $\overline{S}$
6. Run **ImplicitBFS**($(\overline{S} \cup \{\hat{s}, \hat{t}\}, \overline{E})$, $\hat{s}$, $\hat{t}$)
   // **ImplicitBFS** executes in the same way as **BFS**
   // except for the case "$(a, b) \in \overline{E}$?" is queried,

// i.e., it is asked whether G(V' ∪ $\overline{S}$) has an
// edge (a, b)? In this case, the query is answered
// by the following process 6.1 ~ 6.5.

6.1. **For** every x ∈ V
// V_x = the set of vertices of $\underline{G}$(V \ $\overline{S}$)'s
// connected component containing x.

6.2. **If** PlanarReach(G(V_x ∪ $\overline{S}$), a, b, n) is True

6.3. **then** Return True for the query

6.4. **End_For**

6.5. Return False for the query

PlanarReach Algorithm (for directed planar graphs):

Input: A directed planar graph G with vertices s and t, and total number of vertices n.

Goal: To check if there's a path from s to t.

Steps:

1. Count Vertices:
Let $\hat{n}$ be the number of vertices in the graph G.

2. Small Graph Optimization:
If $\hat{n} <= n^{1/2}$:
- Just use BFS (Breadth-First Search) to check for a path from s to t.
- Return the result.

3. Large Graph Case:
- Otherwise, set $\hat{r} = n^{1-\varepsilon}$ (where ε is a small positive constant).

4. Separator Family:
- Compute a "separator family" S to simplify the graph into smaller subproblems.

5. Modified BFS:
- Run a special BFS called "ImplicitBFS" on a modified graph:
- ImplicitBFS works like normal BFS but handles edge queries differently:
- If the BFS queries whether there is an edge between a and b, it uses the separator information to answer the question indirectly.

6. Check Connections for Each Vertex:
For each vertex x in the graph G:
- Define Vx as the connected component of G that contains x, excluding the separator set S.

- Check if s can reach t using PlanarReach within G(Vx ∪ S).
  - If the path exists, return True.

7. No Path Found:
  - If no path is found for any vertex x, return False.

Key Idea:
The algorithm cleverly handles large graphs by breaking them into smaller, manageable pieces (using separator families) and combining the results from these smaller subproblems. It avoids directly handling the entire graph at once for efficiency.

For solving an instance (G, s, t) of the planar graph reachability problem, it suffices to call PlanarReach(G, s, t, n).

Now, we analyze the space and time complexity of this algorithm; let S and T denote its space and time complexity functions. First note that, since $(1 - \epsilon)^k \leq 1/2$ for $k = O(1/\epsilon)$, the depth of recursion is $O(1/\epsilon)$, which is a constant.

We begin with the space complexity. Our goal is to show that $S(n) = \tilde{O}(n^{1/2+\epsilon})$. Consider $S(\hat{n})$ for any $\hat{n} > n^{1/2}$, then algorithm uses space $\tilde{O}(n^{1/2+\epsilon/2})$. Otherwise, It runs ImplicitBFS on $(\overline{S} \cup \{\hat{s}, \hat{t}\}, \overline{E})$, and its main computation can be done by using $\tilde{O}(|\overline{S}|)$ space like the standard BFS. On the other hand, for each query $(a, b) \in \overline{E}$ asked in the computation, we need to run Lines 6.1 ~ 6.5, and the space needed for this computation is essentially $S(|V_x| + |S|) \leq S(2\hat{n}^{1-\epsilon})$ for Line 6.2. Hence we get the following recurrence.

$$\mathcal{S}(\widehat{n}) = \begin{cases} \widetilde{O}\left(\widehat{n}^{1/2+\epsilon/2}\right) + \mathcal{S}\left(2\widehat{n}^{1-\epsilon}\right) & \text{if } \widehat{n} > n^{1/2}, \\ \widetilde{O}\left(n^{1/2}\right) & \text{otherwise.} \end{cases}$$

Since the recursion depth is bounded by $O(1/\epsilon)$, it is easy to see that $S(n) = O(1/\epsilon)\tilde{O}(n^{1/2+\epsilon/2}) = \tilde{O}(n^{1/2+\epsilon/2})$, which is sufficient for our goal.
Next consider the time complexity. We need to be precise only up to a polynomial factor. Then by analysis similar to the above, we have the following recurrence.

$$\mathcal{T}(\widehat{n}) = \begin{cases} q(n)\left(p_1(\widehat{n})\mathcal{T}\left(2\widehat{n}^{1-\epsilon}\right) + p_2(\widehat{n})\right) & \text{if } \widehat{n} > n^{1/2}, \\ q(n)\widetilde{O}\left(n^{1/2}\right) & \text{otherwise.} \end{cases}$$

Here $p_1(\hat{n})$ is the number of times we make the recursive calls of Line 6.2, and $p_2(\hat{n})$ is the time needed by SepFamily at Line 5. On the other hand, a polynomial q(n) is for the overhead when $\hat{G}$ is given implicitly. Again from the $O(1/\epsilon)$ bound for the recursion depth, it is easy to see the bound $T(n) = p(n)^{O(1/\epsilon)}$ holds for some polynomial p(n).

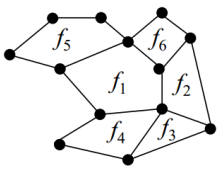
SPACE AND TIME EFFICIENT SEPARATOR ALGORITHM

Theorem 2. *There is an algorithm that takes an undirected planar graph G with n vertices as input and outputs a (8/9)- separator of G. This algorithm runs in polynomial time and uses $\tilde{O}(\sqrt{n})$ space.*

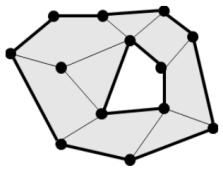
We assume that a given input undirected graph for our separator algorithm is connected and triangulated. In fact, in our application of this algorithm, only a connected graph is given as an input to the algorithm. Also triangulation is easy by adding "imaginary edges" so that every face becomes a triangle. We may use the following rule: For each face, consider the smallest indexed vertex on it. For every other vertex on that face, add an edge to this vertex, thereby triangulating the face. We may assume that this triangulation algorithm is applied before the execution of the separator algorithm.

Preliminaries for the Separator Theorem

Let G be a planar graph (not necessarily triangulated) and let f be a face. The face-size of f is the number of edges (hence, that of vertices) of f. Two faces of G are edge-connected if they share an edge. A set of faces R is edge-connected if for every pair of faces f and g in R, there exist faces $f_1, \cdots, f_i$ in R such that f and f_1 are edge-connected, f_i and g are edge-connected, and f_j and f_{j+1} are edge-connected for all j, $1 \leq j \leq i - 1$. A region of a planar graph is a set of edge-connected faces. The boundary of a region is the set of edges such that each edge lies on exactly one face of the region. Given a region R, we denote the boundary with $\mathcal{B}(R)$. It is known that the boundary of any region can be decomposed into a set of disjoint simple cycles.



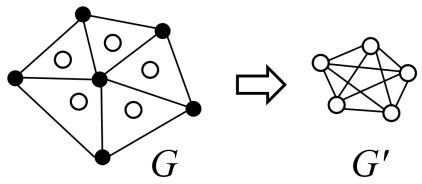
faces f_1, f_2, ...



a shadow part: a region
two bold lines: a boundary

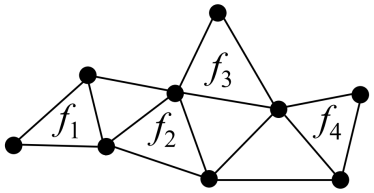
From now on let G denote any planar graph that is connected and triangulated. Below we will usually regard triangular faces as vertices of a related graph called "face-vertex graph", which is different from the standard dual graph. The face-vertex graph of G is a graph denoted as G' = (V',E') where V' is the set of triangle faces of G and E' is the set of pairs (f_1, f_2) of triangle faces of G such

that f_1 and f_2 share a vertex in G. We add prefix "tr-" to distinguish terms for face-vertex graphs. For example, a vertex of G' , which corresponds to some triangular face of G, is called a tr-vertex, and an edge of G' is called as a tr-edge. A path of G' consisting of tr-edges is called a tr-path; we often regard a tr-path as a sequence of faces.



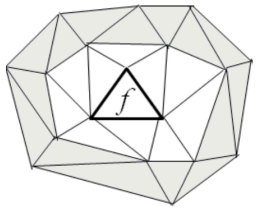
white circles: faces in G and tr-vertices in G'

For any tr-vertices f_1 and f_2 in G' , the distance between f_1 and f_2 (denoted by dist(f_1, f_2)) is the length (i.e., the number of tr-edges) of the shortest tr-path between f_1 and f_2 in G'. Note that for any v_1 and v_2 of G that lie on triangle faces f_1 and f_2 respectively, we have dist(v_1, v_2) ≤ dist(f_1, f_2) + 1.



tr-path (f_1, f_2, f_3, f_4):
a shortest tr-path from f_1 to f_4

For a tr-vertex f and an integer r, let ℓ(f, r) denote the set of tr-vertices that are at distance exactly r from f. For any d ≥ 1, the d-radius ball around f is the set $B_d(f) = \{g \in V' \mid dist(f, g) \le d\}$. Given a tr-vertex f and a region R, the distance between f and R is defined by $dist(f, R) = \min_{g \in R}\{dist(f, g)\}$.



$\ell(f, 2)$

Definition 3. *For any tr-vertex f of G' , the k- neighborhood of f (denoted as $N_k(f)$) is the set of k tr-vertices closest to f with respect to the distance function dist. More formally,*

$$N_k(f) = B_r(f) \cup F$$

where r is the maximum integer such that $|B_r(f)| \le k$, and F is an edge-connected subset of ℓ(f, r + 1) so that $|N_k(f)|$ becomes exactly k.

Definition 4. *A set I of faces (tr-vertices) is a k-maximal independent set if*

- *for every f, g ∈ I, $N_k(f) \cap N_k(g) = \varnothing$, and*
- *for every f ′ ∉ I, there exists a face f ∈ I such that $N_k(f') \cap N_k(f) \neq \varnothing$.*

The size of a k-maximal independent set is O(n/k).

Lemma 4. *There is an algorithm that takes a planar graph G as an input, and outputs a k-maximal independent set in polynomial time and $\tilde{O}(n/k + k)$ space.*

Definition 5. *Let f be a tr-vertex and $N_k(f)$ be its k-neighborhood. Let r_0 be the largest number such that $\ell(f, r_0) \subseteq N_k(f)$ and $|\ell(f, r_0)| \leq \sqrt{k}$. The core of f is defined as the union of $\ell(f, i)$, $1 \leq i \leq r_0$.*

A core is a region.

Lemma 5. *For every tr-vertex f, the following holds:*

- *The size of the boundary of the core of f is at most $\sqrt{k}$.*
- *For every tr-vertex $f' \in N_k(f)$ and not in core of f, there is a tr-vertex g in the core of f such that $dist(f', g) \leq \sqrt{k}$.*

We extend the notion of core of a face to core of a vertex. For any vertex v of G, let f be a triangle face (for consistency we take the lexicographically smallest face) on which v lies. The core of v is simply the core of f.

Next, we define the notion of "Voronoi region". We fix some k-maximal independent set I. For every face g of G, we associate a unique member of I as follows: If $g \in N_k(f)$ for some $f \in I$, then g is associated to f. For $g \notin N_k(f)$ for any $f \in I$, g is associated with the lexicographically first $f \in I$ such that $dist(N_k(f), g)$ is the smallest among all faces in I. The Voronoi region of $f \in I$, denoted as V(f), is the set of faces that are associated with f.

Lemma 6. *Every Voronoi region is edge-connected. The diameter of a Voronoi region is O(k); that is, the distance between every pair of tr-vertices in the Voronoi region is O(k).*

Lemma 7. *There is an algorithm that takes a planar graph G, a tr-vertex g of G′ , and a k-maximal independent set I, as an input, and computes $f \in I$ such that $g \in V(f)$. Moreover, this algorithm runs in polynomial time and $\tilde{O}(k)$ space.*

Lemma 8. *There is a polynomial-time and $\tilde{O}(k)$-space algorithm that, given a planar graph G, a k-maximal independent set I, and $f \in I$, constructs a **BFS** tree of V(f) rooted at f. The diameter of this tree is O(k).*

The Separator Algorithm

Let G be an input planar graph with 'n' vertices. We set k = √n, and first compute a k-maximal independent set I and check whether there is some f ∈ I such that V(f) has more than n/3 vertices of G. If such a Voronoi region exists, then we simply use the algorithm of Lipton and Tarjan to get a (2/3)-separator of V(f), which is also a (8/9)-separator of the original graph G. Since we can construct a **BFS** tree of V (f) in small space (Lemma 8), we can implement the algorithm of Lipton and Tarjan on a Voronoi region using small space in polynomial time.

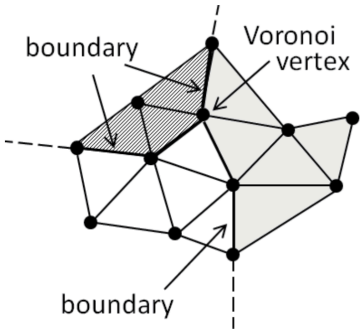
Nontrivial treatment is necessary for the case where all Voronoi regions are small. In this case, we construct a small weighted planar subgraph, H of G (and a planar embedding) with a rational weight assigned to each face where the weights sum to 1. We then compute a weighted separator of H. This weighted separator of H will also be a separator of G.

For any ρ, $0 < \rho < 1$, a subset S of vertices of H is called a ρ-weight-separator if removal of S disconnects H into two components so that each component's total weight is at most ρ. We use the following theorem to construct the weighted separator.

Theorem 9. *There is a polynomial-time and $\tilde{O}(m)$-space algorithm that takes as an input a weighted planar graph H satisfying the following three conditions and outputs a (2/3)-weight-separator of size $O(d\sqrt{m})$: (i) H has m faces, (ii) the maximum face-size is d, and (iii) there is no face with a weight more than 2/3.*

To explain the main idea behind our algorithm, we consider that the boundary of every Voronoi region is one simple cycle.

We need one more notion. For any Voronoi region, consider its boundary and vertices (of G) on the boundary. All such vertices belong to at least two Voronoi regions. On the other hand, there may be some vertex that belongs to three or more triangle faces each of which belongs to a different Voronoi region; we call such vertices **Voronoi vertices**. The other vertices (on the boundary) are called **non Voronoi vertices**.



We now establish our main lemma which will be used in the proof of Theorem 2.

Lemma 10. *There is a polynomial-time, $\tilde{O}(n/k + k)$-space algorithm, that takes a planar graph $G = (V, E)$ (with n vertices) as input and outputs either*

1) *a Voronoi region V(f) such that the number of vertices in V(f) is at least n/3.*
   or
2) *a weighted planar subgraph H of G with the following properties:*
   a) Every weight in H is less than 2/3.
   b) The number of faces in H is O(n/k).
   c) The size of each face of H is $O(\sqrt{k})$.
   d) Any weight-separator of H is a separator of G.

Proof:

The algorithm first computes a k-maximal independent set I using Lemma 4, and stores this set in the memory. This takes $\tilde{O}(n/k+k)$ space and polynomial-time. For each $f \in I$, count the number of vertices in V(f) as follows: Initialize a counter to zero. Consider a vertex $v \in V$. By cycling through all faces on which v lies, using Lemma 7, check if some face belongs to V(f). If some face belongs to V(f), then increment the counter. If the counter reaches n/3, then return V(f). Checking whether a face belongs to V(f) or not can be done in $\tilde{O}(k)$ space and polynomial-time due to Lemma 7. Thus the total time taken to output V(f) is polynomial and the space is $\tilde{O}(n/k + k)$.

Assume that for every $f \in I$, the number of vertices in V(f) < n/3. Now we define our weighted planar graph H. We first describe the graph part, and later describe the weights. It is essentially a subgraph of G = (V, E) consisting of a subset of edges of E (and their adjacent vertices). For a given G and its k-maximal independent set I, our algorithm executes the following three sub steps.

(1) For each $f \in I$, output the boundary of the core of f.

(2) For every $v \in V$, determine if it is a Voronoi vertex. If v is a Voronoi vertex, then output the boundary of the core of v.

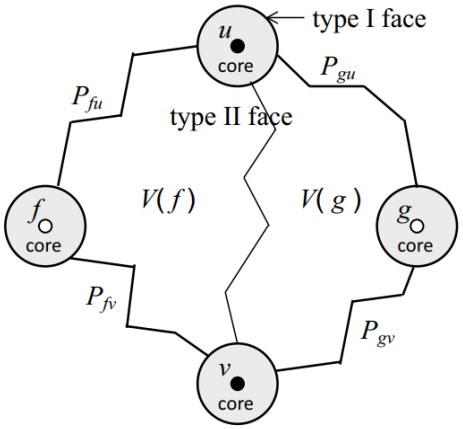
(3) For every pair f and g of tr-vertices in I such that $\mathcal{B}(V(f))$ and $\mathcal{B}(V(g))$ intersect, do the following: Compute all Voronoi vertices that are common to $\mathcal{B}(V(f))$ and $\mathcal{B}(V(g))$. For every such vertex v, compute a path $\hat{P}_{f,v}$ from f to v based on the tr-path in the **BFS** tree of V(f). Then select the part of $\hat{P}_{f,v}$ that lies outside of the cores of f and v, and output it as $\hat{P}_{f,v}$. Similarly, output $\hat{P}_{g,v}$.

Next we specify the way to assign weights to the faces of H. For each face h of H, assign a weight n_h/n, where n_h is the number of vertices of G that lie inside of h so that vertices of H are counted once at some face. Now it is clear that H is planar and the sum of all weights is 1. Also from the above way of assigning weights it is clear that any ρ-weight-separator of H is an ρ-separator of G.

Next we examine the faces of H and their size parameters. Note that a face is defined by edges and that every edge of H is either a part of the boundary of some core or a part of path $\hat{P}_{f,v}$ for some $f \in I$ and some Voronoi vertex v. We can classify all faces to the following two types.

Type I: A face consisting of edges from the boundary of some core that are produced by sub steps (1) or (2).

Type II: A face consisting of the edges of four paths $\hat{P}_{f,u}$, $\hat{P}_{f,v}$, $\hat{P}_{g,u}$, $\hat{P}_{g,v}$ and some edges in boundaries of the cores of f, g, u, and v, where f and g are tr-vertices of G such that $\mathcal{B}$(V(f)) and $\mathcal{B}$(V(g)) intersect, and u and v are Voronoi vertices that appear in both V(f) and V(g). Note that the edges belonging to these paths are produced by sub step (3) and the edges from the cores are produced by sub steps (1) or (2).



Claim 1. *All faces of H are either* type I *or* type II.

Claim 2. *The face-size of all faces of H is O(√k).*

Claim 3. *The number of faces of H is O(n/k).*

Claim 4. *If there is no f ∈ I such that V(f) contains more than n/3 vertices of G, then H has no face with weight > 2/3.*

Claim 5. *The above procedure for computing H can be implemented in polynomial time and Õ(n/k + k) space.*

Now we are ready to present the proof of Theorem 2.

Proof of Theorem 2: We explain the execution of our separator algorithm Separator on a given planar graph G of n vertices. As discussed above, we may assume that G is connected and triangulated and that some combinatorial embedding is also given as an input.

The algorithm sets k = √n and runs the algorithm from Lemma 10. If the algorithm outputs a big Voronoi region V(f) with more than n/3 vertices, then (implicitly) construct a BFS tree of V(f) using

the algorithm from Lemma 8. The **BFS** tree has diameter $O(\sqrt{n})$. Let $n' \geq n/3$ denote the number of vertices of G in V(f). We now apply the algorithm of Lipton and Tarjan on the subgraph induced by V(f). Given a **BFS** tree (with diameter d) of the input graph, Lipton and Tarjan's algorithm can be implemented in linear time and logarithmic space to compute a separator of size O(d). Since the **BFS** of V(f) has diameter $O(\sqrt{n})$, and can be computed in polynomial-time using $\tilde{O}(\sqrt{n})$- space, we can compute a (2/3)-separator of V(f) using space $O(\sqrt{n})$ and polynomial-time. Certainly, this separator separates some subgraph of G of size $\geq n'/3$ ($\geq n/9$) from not only V(f) but also from G; thus, it is also an (8/9)- separator, satisfying the theorem.

If no big Voronoi region exists, the algorithm produces subgraph H of G. Now apply Theorem 9 to compute a (2/3)-weight-separator of H. From Lemma 10 we know that H has $O(n/k)$ ($= O(\sqrt{n})$) faces with their face-size bounded by $O(\sqrt{k})$ ($= O(n^{1/4})$). Thus, Theorem 9 runs in polynomial time and $\tilde{O}(\sqrt{n})$ space, and the size of the separator is $O(\sqrt{n})$. On the other hand, any 2/3-weighted separator for H is also a 2/3- separator for G. Therefore, both the output and the efficiency of the algorithm satisfy the required conditions.

A SUBLINEAR SPACE, POLYNOMIAL TIME ALGORITHM FOR DIRECTED s-t CONNECTIVITY

ABSTRACT

Directed s-t connectivity is the problem of detecting whether there is a path from vertex s to vertex t in a directed graph. We present the first known deterministic sublinear space, polynomial time algorithm for directed s-t connectivity. For n-vertex graphs, our algorithm can use as little as $n/2^{\Theta(\sqrt{\log n})}$ *space while still running in polynomial time.*

INTRODUCTION

The s-t connectivity problem, detecting whether there is a path from a distinguished vertex s to a distinguished vertex t in a directed graph, is a fundamental one, since it is the natural abstraction of many computational search processes, and a basic building block for more complex graph algorithms. In particular, the s-t connectivity problem for directed graphs (STCON) is the prototypical complete problem for non-deterministic logarithmic space. Both STCON and the undirected version of the problem, USTCON, are DLOG-hard—any problem solvable deterministically in logarithmic space can be reduced to either problem.

The standard algorithms for connectivity, breadth- and depth-first search, run in optimal time $\Theta(m+n)$ and use $\Theta(n \log n)$ space. At the other extreme, Savitch's theorem provides a small space ($\Theta(\log^2 n)$) algorithm that requires exponential time ($n^{\Theta(\log n)}$) in its space bound.

Prior to this paper, there was no corresponding sublinear space, polynomial time algorithm known for STCON, and there was some evidence suggesting that none was possible. It has been conjectured that no deterministic STCON algorithm can run in simultaneous polynomial time and polylogarithmic space. Tompa shows that certain natural approaches to solving STCON admit no such solution. Indeed, he shows that for these approaches, performance degrades sharply with decreasing space. Space O(n) implies superpolynomial time, and space $n^{1-\epsilon}$ for fixed $\epsilon > 0$ implies time $n^{\Omega(\log n)}$, essentially as slow as Savitch's algorithm.

The main result of this paper is a new deterministic algorithm for directed s-t connectivity that achieves polynomial time and sublinear space simultaneously. The algorithm can use as little as $n/2^{\Theta(\sqrt{\log n})}$ *space while still running in polynomial time.* As part of this algorithm, we present an algorithm that finds short paths in a directed graph in polynomial time and sublinear space. Interestingly, the algorithm for the short paths problem is a generalization of two well-known algorithms for STCON. In one extreme it reduces to a variant of the linear time breadth-first search algorithm, and in the other extreme it reduces to the $O(\log^2 n)$ space, superpolynomial time algorithm of Savitch.

The algorithm to solve STCON in polynomial time and sublinear space is constructed from two algorithms with different time-space tradeoffs. The first performs a modified breadth-first search of the graph, while the second finds short paths. Alone, neither algorithm can solve STCON in simultaneous polynomial time and sublinear space. In the following sections, we present the breadth-first search algorithm, short paths algorithm and show how these two algorithms can be combined to yield the desired result.

THE BREADTH-FIRST SEARCH TRADEOFF

Consider the tree constructed by a breadth-first search beginning at s. The tree can contain n vertices and thus requires O(n log n) space to store. Instead of constructing the entire tree, our modified breadth-first search generates a fraction of the tree.

Suppose we want our modified tree to contain at most n/λ vertices. We can do this by only storing the vertices in every λth level of the tree. Number the levels of the tree 0, 1, ..., $n-1$, where a vertex v is on level ℓ if the shortest path from s to v is of length ℓ. Divide the levels into equivalence classes C_0, C_1,...,$C_{\lambda-1}$ based on their number mod λ. Besides s, the algorithm stores only the vertices in one equivalence class C_j , where j is the smallest value for which C_j has no more than the average number of vertices, n/λ.

The algorithm constructs this partial tree one level at a time. It begins with level 0, which consists of s only, and generates levels $j, j + \lambda, j + 2\lambda, ..., j + \lambda \cdot \lfloor n/\lambda \rfloor$. Given a set, S, of vertices, we can find all vertices within distance λ of S in time $n^{O(\lambda)}$ and space $O(\lambda \log n)$ by enumerating all possible paths of length at most λ and checking which paths exist in G. This can be used to generate the levels of the partial tree. Let V_i be the vertices in levels $0, j, j+\lambda,..., j+i\lambda$. Consider the set of vertices, U, that are within distance λ of a vertex in V_i. Clearly, U contains all the vertices in level $j + (i + 1)\lambda$. However, U may also contain vertices in lower numbered levels. The vertices in level $j + (i + 1)\lambda$ are those vertices in U that are not within distance $\lambda - 1$ of a vertex in V_i. Thus, to get V_{i+1} we add to V_i all vertices that are within distance λ but not $\lambda - 1$ of V_i.

Note that to find an equivalence class with at most n/λ vertices, the algorithm just tries all classes in order, discarding a class if it generates too many vertices.

The algorithm's space bound is dominated by the number of vertices in S and S' , and the space needed to test whether a vertex is within distance λ of a vertex in S. There are never more than $n/\lambda + 1$ vertices in S and S' , so the algorithm uses $O((n \log n)/\lambda)$ space to store these vertices. The time bound is dominated by repeatedly testing whether a vertex is within distance λ of a vertex in S. This test is performed $O(n^3/\lambda)$ times—the innermost loop to find the vertices on the next level of the tree makes $O(n \cdot n/\lambda)$ such tests (testing for a path from the $O(n/\lambda)$ vertices in S to all other O(n) vertices), and is executed $O(\lambda \cdot n/\lambda)$ times.

In summary, we have shown the following.

Theorem 2.1. *For any n-vertex directed graph and any integer λ, $1 \leq \lambda \leq n$, the breadth-first search algorithm presented above solves s-t connectivity in space $O((n \log n)/\lambda + S_{PATH}(\lambda, n))$ and time $O((n^3/\lambda) \cdot T_{PATH}(\lambda, n))$, where $S_{PATH}(\lambda, n)$ and $T_{PATH}(\lambda, n)$ denote the space and time bounds, respectively, of the algorithm used to test for a path of length at most λ between two vertices in an n-vertex graph.*

Note that we assume that testing for a path of length at most j, j − 1, or λ − 1 will not take asymptotically more time or space than testing for a path of length at most λ.

Breadth-first search algorithm pseudocode

```
Algorithm Bfs (integer: λ);
                                        {remember every λth level of the breadth-first search tree}
    for j = 0 to λ - 1 do begin          {first level to remember, apart from level 0}
      S = {s}.
      for all vertices, v do begin
        if v within distance j of s and
          v not within distance j - 1 of s then
            if |S| > n / λ then try next j.     {Don't store more than n/λ vertices, + vertex s}
            else add v to S.
      end;
      for i = 1 to ⌊n/λ⌋ do begin
        S' = ∅.
        for all vertices, v do begin                {Find vertices on the next level. *}
          if  v within distance λ of some vertex in S and
              v not within distance λ - 1 of any vertex in S then
              if |S| + |S'| > n / λ then try next j.
              else add v to S'.
        end;
        S = S ∪ S'.
      end;
      if t within distance λ of a vertex in S then return (CONNECTED);
      else return (NOT CONNECTED);
    end;
end Bfs.
```

Using a straightforward enumeration of all paths, testing whether a vertex is within distance λ requires $n^{O(\lambda)}$ time and O(λ log n) space. This algorithm is not sufficient for our purposes. In particular, if λ is asymptotically greater than a constant, the algorithm uses superpolynomial time. If we restrict our input to graphs with bounded degree, there is a slight improvement. In a graph where the outdegree is bounded by d, the number of paths of length λ from a vertex is at most d^{λ} . For these graphs, λ can be O(log n) and the algorithm will run in polynomial time. Note that the overall algorithm still does not use sublinear space in this case, even though the subroutine for finding paths of length λ does.
The problem with this algorithm is its method of finding vertices within distance λ. Explicitly enumerating all paths is not very clever, and uses too much time. There is hope for improvement, though, since this method uses only O(λ log n) space, much less than the O(nλ log n) used by the rest of the algorithm. Indeed, in the next section we give an algorithm that uses more space but runs much faster.

THE SHORT PATHS TRADEOFF
Consider the bounded s-t connectivity problem (bounded STCON).

Definition 3.1. *For any real-valued function f(n), the f(n)-bounded s-t connectivity problem is, given an n-vertex directed graph G and two distinguished vertices s and t, to determine whether there is a path in G from s to t of length less than or equal to f(n).*

Solving bounded STCON for general f(n) is at least as hard as solving STCON. We are interested in the short paths problem, informally defined as the bounded STCON problem where f(n) is small. The short paths problem is a special case of STCON that seems to retain many of the difficulties of the general problem. It is particularly interesting given the breadth-first search algorithm above, because a more efficient method of finding short paths would clearly lead to an improvement in that algorithm's time bound.
Our second tradeoff is an algorithm that solves the short paths problem for many f(n) in sublinear space and polynomial time. As will become clear, we will eventually want $f(n)=2^{\Theta(\sqrt{\log n})}$, but to simplify the following discussion, we begin with the more modest goal of finding a sublinear space, polynomial time algorithm for the short paths problem with $f(n) = \log^c n$, for some integer constant $c \geq 1$.

Suppose we divide the vertices into k sets, according to their vertex number mod k. Then, every path of length L (L = f(n)) can be mapped to an (L+1)-digit number in base k, where digit i has value j if and only if the ith vertex in the path is in set j. Conversely, each such number defines a set of possible paths of length L.
Given this mapping, the algorithm is straightforward: generate all possible (L+1)- digit k-ary numbers, and check for each number whether there is a path in the graph that matches it. For a given k-ary number, the algorithm uses approximately 2n/k space to test for the existence of a matching path in the graph, as follows. Suppose we are looking for a path from s to t and want to

test the (L + 1)-digit number $\langle s \bmod k, d_1, d_2, \ldots, d_{L-1}, t \bmod k. \rangle$. We begin with a bit vector of size $\lceil n/k \rceil$, which corresponds to the vertex set d_1. Zero the vector, and then examine the outedges of s, marking any vertex v in set d_1 (by setting the corresponding bit in the vector) if we find an edge from s to v. When we are finished, the marked vertices in the vector are the vertices in d_1 that have a path from s that maps to the first two digits of the number. Using this vector, we can run a similar process to find the vertices in d_2 that have a path from s that maps to the first three digits of the number, and store them in a second vector of size $\lceil n/k \rceil$. In general, given a bit vector of length $\lceil n/k \rceil$ representing the vertices in d_i with a path from s that maps to the first i + 1 digits of the number, we use the other vector to store the vertices in d_{i+1} with a path from s that maps to the first i+2 digits.
Notice that the algorithm as given does not solve the short paths problem, as it tests for the existence of a path from s to t of length exactly L, not at most L.

Short paths algorithm pseudocode

```
Algorithm SP (integer: k, L; vertex s, t);
                                   {Test for a path of length L between s and t using space O(n/k)}
    Create V0 and V1, two ⌈n/k⌉ bit vectors.
    for all (L + 1)-digit numbers in base k,
      ⟨d0 = s mod k, d1, ..., dL-1, dL = t mod k⟩ do begin
        Set all bits in V0 to zero, and mark s (set the corresponding bit to 1).
        for i = 1 to L do begin
          Set all bits in V(i mod 2) to zero.
                                                  {Find edges from d(i-1) to di}
          for all u in d(i-1) marked in V((i-1) mod 2) and all v in di do begin
            if (u, v) is an edge then
            mark v in V(i mod 2).
          end;
        end;
        if t is marked in V(L mod 2) then return (CONNECTED);
    end;
    return (NOT CONNECTED);
end SP.
```

The algorithm uses space O(n/k) to store the vectors, and O(L log k) to write down the path to be tested. Let D be the maximum number of edges from one set of vertices d_i to another set of vertices d_j (i and j can be the same). For all steps in each path, we do at most O(n/k + D) work zeroing the vector and testing for edges from d_{i-1} to d_i. Since $D = O(n^2/k^2)$, the algorithm uses
$O(k^L L \cdot n^2/k^2) = O(k^L n^3)$ time to test all L steps on each of the k^L paths.

Unfortunately, this does not reach our goal of polynomial time and sublinear space when $L = \log^c n$. With a distance as small as log n, k^L is only polynomial if k is constant, and if k is constant, the algorithm does not use sublinear space. We can achieve polynomial time and sublinear space by reducing the distance the algorithm searches. For example, if $L = (\log n)/ \log \log n$, k can be $\log^c n$ for any constant c, and the algorithm will run in $O(n/ \log^c n)$ space and
$O((\log n)^{c \log n/ \log \log n} n^3) = O(n^{c+3})$ time.

The algorithm can be improved by invoking it recursively. Consider the loop in the algorithm that tests for edges between one set of vertices and the next. This loop, in effect, finds paths of length one from marked vertices in the first set to vertices in the second set. Instead of finding paths of length one, we can use the short paths algorithm to find paths of length L, yielding an algorithm that uses twice as much space, but finds paths of length L^2. In general, using $r \geq 1$ levels of recursion, the improved algorithm can find paths of length L^r using $O(r(n/k + L \log k))$ space. If we make a recursive call for every possible pair of vertices in $d_{i-1} \times d_i$, we get a time bound of $O((k^L L \cdot n^2/k^2)^r) = O(n^{2r+1}k^{rL})$, since $L^r = O(n)$. This algorithm uses a further refinement to improve the time bound—one recursive call is used to find all vertices in d_i reachable from any reachable vertex in d_{i-1}.
Given the discussion above, the time used by the recursive algorithm is bounded by the following recurrence relation, where T(j) is the time used by the algorithm with j levels of recursion. For an appropriately chosen constant c,

$$T(j) = \begin{cases} O(n^2/k^2) & \text{if } j = 0, \\ k^L L(T(j-1) + cn/k) & \text{if } j > 0. \end{cases}$$

In the base case, the algorithm does $O(n^2/k^2)$ work. At other levels, the algorithm makes $k^L L$ recursive calls to itself, as well as doing some auxiliary work, such as setting all vector entries to zero. Solving the recurrence relation for j = r gives time $O((k^L L)^r \cdot n^2/k^2) = O(n^3 k^{rL})$.

Recursive short paths algorithm pseudocode

```
Algorithm SPR (integer: k, L, r, d_s, d_t; vector V_s): vector;
                        {Return the vector of vertices in set dt that are reachable by paths
                          of length L^r from vertices in set d_s that are marked in vector V_s}
    Create V_0, V_1, and V_t, three ⌈n/k⌉-bit vectors. Set all bits in V_t to zero.
    if r = 0 then                                                          {base case}
      for all u in d_s marked in V_s and all v in d_t do begin
        if (u, v) is an edge then
          mark v in V_t.
      end;
```

```
    else
      for all (L + 1)-digit numbers in base k,
          ⟨d0 = ds, d1, ..., dL-1, dL = dt⟩ do begin
          V0 = Vs .
          for i = 1 to L do begin                    {Find paths from di-1 to di}
              Vi mod 2 = SPR(k, L, r - 1, di-1, di, V(i-1) mod 2).
          end;
          Set all bits in Vt that are set in VL mod 2 .
      end;
    return (Vt);
end SPR.
```

In summary, we have shown the following.

Theorem 3.2. *For arbitrary integers r, k, and L, such that $r \geq 1$, $L \geq 1$, $n \geq k \geq 1$, and $L^r \leq n$, the recursive short paths algorithm, presented above, can search to distance L^r in time $O(k^{rL}L^r \cdot n^2/k^2)$ $(= O(n^3k^{rL}))$ and space $O(r(n/k + L\log k))$.*

This recursive algorithm meets our goal of finding a sublinear space, polynomial time algorithm that detects paths of polylogarithmic length. For example, for $L = \log n/ \log\log n$, $k = \log^r n$, and constant $r \geq 2$, the algorithm searches to distance $L^r = \omega(\log^{r-1} n)$ in time $O(n^3k^{rL}) = O(n^{r2+3})$ and space $O(rn/ \log^r n)$. However, as mentioned in the introduction, this algorithm does not by itself give a polynomial time, sublinear space algorithm for STCON. The algorithm searches to distance L^r by testing k^{rL} numbers. If $L^r = n$, then k^{rL} is polynomial only if $k = O(1)$. But if $k = O(1)$, the algorithm does not use sublinear space.

The algorithm, which was designed to solve the short paths problem, actually solves bounded STCON, and is thus a general algorithm for s-t connectivity.

COMBINING THE TWO ALGORITHMS

As an immediate consequence of the previous two sections, we have an algorithm for STCON using sublinear space and polynomial time: use the modified breadth-first search algorithm to find every $(\log^c n)$th level of the tree (for integer constant $c \geq 2$), with the recursive short paths algorithm as a subroutine to find the paths between levels.

In general, if we set λ in the breadth-first search algorithm to be L^r, the breadth-first search algorithm finds every (L^r)th level of the tree, and the short paths algorithm searches to distance L^r. Substituting the space bound for the short paths algorithm for the term $S_{PATH}(\lambda, n)$ in the breadth-first search algorithm, we get a space bound for this algorithm of

(4.1) $O((n \log n)/L^r + r(n/k + L\log k))$,

where the first term corresponds to the space used by the partial breadth-first tree, and the second to the space used to find short paths. Substituting the short paths time bound for the term $T_{PATH}(\lambda, n)$ in the breadth-first search time bound gives a time bound of

$$O((n^3/L^r) \cdot k^{rL}L^r \cdot n^2/k^2) = O(n^5k^{rL-2}).$$

Combining the two algorithms efficiently

```
                                  {S is the set of vertices on previous tree levels. S' (initially
                                   the empty set) will be the set of vertices on the next level}
for i1 = 0 to k - 1 do begin
   Si1 = {all vertices whose vertex number mod k = i1}.
   P = ∅.                                  {P will be all vertices in Si1 on the next tree level}
   for i2 = 0 to k - 1 do begin
      Si2 = {all vertices whose vertex number mod k = i2}.
      Q = S ∩ Si2.                         {Q is all vertices in Si2 on previous tree levels}
      A = {all vertices in Si1 within distance L^r of a vertex in Q}.
      B = {all vertices in Si1 within distance L^r - 1 of a vertex in Q}.
      P = P ∪ (A - B).
   end;
   if |S| + |S' ∪ P| > n/L^r then try next j.
   else S' = S' ∪ P.
end;
```

The above time bound applies when we call the short paths algorithm every time the breadth-first search algorithm needs to know whether one vertex is within distance L^r of another. The two algorithms can be combined more efficiently by noticing that the short paths algorithm can answer many short paths queries in one call; for any pair of sets, (Q, R), such that R is one of the k sets of vertices in the short paths algorithm and Q is a subset of one of the k sets, one call to the short paths algorithm can be used to find all vertices in R within distance L^r of a vertex in Q. Thus, the short paths algorithm only needs to be called $2k^2$ times to generate the next level of the tree, twice for each possible pair of the k sets in the short paths algorithm. Above code should be used in place of the loop in `Bfs` pseudocode (marked with a *) that finds vertices on the next level of the breadth-first search tree. Similar code should replace the earlier loop in `Bfs` pseudocode that finds the vertices on the first level.

The improved version makes a total of $O(k^2n/L^r)$ calls to the short paths algorithm, for a time bound of

(4.2) $$O((k^2n/L^r) \cdot k^{rL}L^r \cdot n^2/k^2) = O(n^3k^{rL})$$

We want to find the minimum amount of space the algorithm can use while still maintaining a polynomial running time. To maintain polynomial time, we must have, for some constant a,

(4.3) $$k^{rL} = n^a$$

For simplicity, we bound expression (4.1) from below by

(4.4) $$\Omega(n/L^r + n/k)$$

(That is, we omit the log n factor in the first summand and the r factor in the second summand, and leave out the third summand altogether.) The minimum value of the bound (4.4) is reached when the denominators are equal. For any given k, the product rL is fixed; thus the quantity L^r reaches its maximum, and the bound reaches its minimum, when L is a constant. Substituting L^r for k in (4.3) and solving for r yields $r = \sqrt{((a/L)\log_L n)} = \Theta(\sqrt{\log n})$, and thus $k = 2^{\Theta(\sqrt{\log n})}$.

Substituting these values, $\sqrt{\log n}$ for r, $2^{\Theta(\sqrt{\log n})}$ for k, and a constant for L, into the simplified space bound expression (4.4) gives a bound of $n/2^{\Theta(\sqrt{\log n})}$. Substituting these same values into the actual space bound expression (4.1) yields the same asymptotic space bound, $n/2^{\Theta(\sqrt{\log n})}$. Since this matches the minimum for the simplified expression, which was a lower bound for this expression, we cannot do any better, and this must be the minimum space bound for the algorithm when using polynomial time.

The results of this section are summarized in the following theorem and its corollary.

Theorem 4.1. *The combined algorithm, described above, solves STCON in space $O((n \log n)/L^r + r(n/k + L\log k))$ and time $O(n^3k^{rL})$, for any integers r, k, and L that satisfy $n \geq k \geq 1$, $r \geq 1$, $L \geq 1$, and $L^r \leq n$.*

Corollary 4.2. *The combined algorithm can solve STCON in time $n^{O(1)}$ and space $n/2^{\Theta(\sqrt{\log n})}$.*
Proof. Choose $r = \sqrt{\log n}$, $k = 2^{\Theta(\sqrt{\log n})}$, and L = 2 in Theorem 4.1. As discussed above, these choices minimize space while retaining polynomial time.

CONCLUSION
Letting L = 2 and $k = 2^r$, we obtain the following corollary.

Corollary 5.1. *The combined algorithm can solve STCON using time $2^{O(\log^2(n/S))} \cdot n^3$ given space S.*

Simultaneous Time-Space Upper Bounds for Red-Blue Path Problem in Planar DAGs★

ABSTRACT

In this paper, we consider the **RedBluePath** *problem, which states that given a graph G whose edges are colored either red or blue and two fixed vertices s and t in G, is there a path from s to t in G that alternates between red and blue edges. The* **RedBluePath** *problem in planar DAGs is* **NL***-complete. We exhibit a polynomial time and* $O(n^{1/2+\epsilon})$ *space algorithm (for any* $\epsilon > 0$*) for the* **RedBluePath** *problem in planar DAG. We also consider a natural relaxation of* **RedBluePath** *problem, denoted as* **EvenPath** *problem. The* **EvenPath** *problem in DAGs is known to be* **NL***-complete. We provide a polynomial time and* $O(n^{1/2+\epsilon})$ *space (for any* $\epsilon > 0$*) bound for* **EvenPath** *problem in planar DAGs.*

INTRODUCTION

A fundamental problem in computer science is the problem of deciding reachability between two vertices in a directed graph. This problem characterizes the complexity class non-deterministic logspace (NL) and hence is an important problem in computational complexity theory. Polynomial time algorithms such as Breadth First Search (BFS) algorithm and Depth First Search (DFS) give a solution to this problem, however they require linear space as well. On the other hand, Savitch showed that reachability can be solved by an $O(\log^2 n)$ space algorithm, however that takes $\Theta(n^{\log n})$ time [1].

It is an important open question whether these two bounds can be achieved by a single algorithm. In other words can we exhibit a polynomial time and $O(\log^k n)$ space algorithm for the reachability problem in directed graphs. Some progress was made in this direction by the first polynomial time and sub-linear space algorithm. It showed that directed reachability can be solved by an $O(n/2^{k\sqrt{\log n}})$ space and polynomial time algorithm [3], by cleverly combining BFS and Savitch's algorithm. Till now this is the best known simultaneous time-space bound known for the directed reachability problem in this direction. [4] This bound has improved for the class of directed planar graphs by a polynomial time and $O(n^{1/2+\epsilon})$ space algorithm by efficiently constructing a planar separator and applying a divide and conquer strategy.

An interesting generalization of the reachability problem is the RedBluePath problem. Given a directed graph where each edge is colored either Red or Blue, the problem is to decide if there is a (simple) directed path between two specified vertices that alternates between red and blue edges. Kulkarni showed that the RedBluePath problem is NL-complete even when restricted to planar DAGs [6]. Unfortunately, no sublinear ($O(n^{1-\epsilon})$, for any $\epsilon > 0$) space and polynomial time algorithm is known for RedBluePath problem in planar DAGs.

A natural relaxation is the EvenPath problem, which asks if there is a (simple) directed path of even length between two specified vertices in a given directed graph. In general, EvenPath problem is NP-complete, but for planar graphs, it is known to be in P. It is also known that for DAGs, this problem is NL-complete. Datta et. al. showed that for planar DAGs, EvenPath problem is in UL. However, no sublinear ($O(n^{1-\epsilon})$, for any $\epsilon > 0$) space and polynomial time algorithm is known for this problem also.

This paper provides a sublinear space and polynomial time bound for both the RedBluePath and EvenPath problem. Our main idea is the use of a space efficient construction of separators for planar graphs. We then devise a modified DFS approach on a smaller graph to solve the RedBluePath problem. As a consequence, we show that a similar bound exists for the directed reachability problem for a class of graphs that is a superset of planar graphs. Using a similar approach, we design an algorithm to detect the presence of an odd length cycle in a directed planar graph, which serves as a building block to solve the EvenPath problem.

CONTRIBUTION OF PAPER

The first contribution of this paper is to give an improved simultaneous timespace bound for the RedBluePath problem in planar DAGs.

Theorem 1. *For any constant $0 < \epsilon < 1/2$, there is an algorithm that solves RedBluePath problem in planar DAGs in polynomial time and $O(n^{1/2+\epsilon})$ space.*

We first construct a separator for the underlying undirected graph and perform a DFS-like search on the separator vertices. Using the reduction given in [6] and the algorithm stated in the above theorem, we get an algorithm to solve the directed reachability problem for a fairly large class of graphs as described in Section 3, that takes polynomial time and $O(n^{1/2+\epsilon})$ space. Thus we are able to beat the BBRS bound for such a class of graphs. One such class is all k-planar graphs, where $k = O(\log^c n)$, for some constant c and this is a strict superset of the set of planar graphs.

In this paper, we also establish a relation between EvenPath problem in a planar DAG and the problem of finding odd length cycle in a directed planar graph and thus we argue that both of these problems have the same simultaneous time-space complexity. We use two colors Red and Blue to color the vertices of the given graph and then use the color assigned to the vertices of the separator to detect the odd length cycle. The conflicting assignment of color to the same vertex in the separator will lead to the presence of an odd length cycle. Here also we use the recursive approach to color the vertices and as a base case we use BFS to solve the problem of detecting odd length cycle in each small component. Thus we have the following result regarding the EvenPath problem.

Theorem 2. *For any constant $0 < \epsilon < 1/2$, there is an algorithm that solves* ***EvenPath*** *problem in planar DAGs in polynomial time and $O(n^{1/2+\epsilon})$ space.*

PRELIMINARIES

A graph G = (V, E) consists of a set of vertices V and a set of edges E where each edge can be represented as an ordered pair (u, v) in case of directed graph and as an unordered pair {u, v} in case of undirected graph, such that $u, v \in V$. Unless otherwise specified, G will denote a directed graph, where |V| = n. Given a graph G and a subset of vertices X, G[X] denotes the subgraph of G induced by X and V(G) denotes the set of vertices present in the graph G. Given a directed graph G, we denote the underlying undirected graph by $\hat{G}$. Throughout this paper, by $\tilde{O}(s(n))$, we mean $O(s(n)(\log n)^{O(1)})$.
The notions of separator and separator family defined below are crucial in this paper.

Definition 1. *A subset of vertices S of an undirected graph G is said to be a ρ-separator (for any constant ρ, $0 < \rho < 1$) if removal of S disconnects G into two sub-graphs A and B such that $|A|, |B| \leq \rho n$ and the size of the separator is the number of vertices in S.*

A subset of vertices $\overline{S}$ of an undirected graph G with n vertices is said to be a r(n)-separator family if the removal of $\overline{S}$ disconnects G into sub-graphs containing at most r(n) vertices.

Now we restate the results and the main tools used in [4] to solve directed planar reachability problem and these results are extensively used in this paper. In [4], the authors construct a 8/9 -separator for a given undirected planar graph.

Theorem 3 ([4]). *(a) Given an undirected planar graph G with n vertices, there is an algorithm* ***PlanarSeparator*** *that outputs a 8/9 -separator of G in polynomial time and $\tilde{O}(\sqrt{n})$ space.*
(b) For any $0 < \epsilon < 1/2$, there is an algorithm ***PlanarSeparatorFamily*** *that takes an undirected planar graph as input and outputs a $n^{1-\epsilon}$-separator family of size $O(n^{1/2+\epsilon})$ in polynomial time and $\tilde{O}(n^{1/2+\epsilon})$ space.*

In [4], the above theorem was used to obtain a new algorithm for reachability in directed planar graph.

Theorem 4 ([4]). For any constant $0 < \epsilon < 1/2$, there is an algorithm DirectedPlanarReach that, given a directed planar graph G and two vertices s and t, decides whether there is a path from s to t. This algorithm runs in time $n^{O(1/\epsilon)}$ and uses $O(n^{1/2+\epsilon})$ space, where n is the number of vertices of G.

RED-BLUE PATH PROBLEM

Deciding Red-Blue Path in Planar DAGs

Given a directed graph G with each edge colored either Red or Blue and two vertices s and t, a red-blue path denotes a path that alternate between Red and Blue edges and the RedBluePath problem decides whether there exists a directed red-blue path from s to t such that the first edge is Red and last edge is Blue. The RedBluePath problem is a generalization of the reachability problem in graphs, however this problem is NL-complete even when restricted to planar DAGs [6].

Proof (of Theorem 1). Consider a planar DAG G. Let $\overline{S}$ be the $n^{(1-\epsilon)}$ -separator family computed by PlanarSeparatorFamily on $\hat{G}$ and let $S = \overline{S} \cup \{s, t\}$. For the sake of convenience, we associate two numerical values to the edge colors – 0 to Red and 1 to Blue. We run the subroutine RedBluePathDetect (Algorithm 3) with the input (G, s, t, n, 0, 1) and if the returned value is true, then there is a directed red-blue path from s to t such that the first edge is Red and last one is Blue. In Algorithm 2, we use the notation $(u, v) \in^{(init,temp)} \overline{E}'$ to decide whether there is a red-blue path from u to v that starts with an edge of color value init and ends with an edge of color value temp.

In Algorithm 1, we use general DFS type search to check the presence of a red-blue path between any two given vertices s' and t'. The only difference with DFS search is that here we explore edges such that the color of the edges alternates between red and blue. If we start from a vertex s', then the for loop (Lines 3 – 8) explore the path starting from s' such that first edge of the path is of specified color. In the main algorithm (Algorithm 3), we use Algorithm 1 as a base case, i.e., when the input graph is small in size (is of size $n^{1/2}$). Otherwise, we first compute S and then run Algorithm 2 on the auxiliary graph $\overline{G} = (S, \overline{E})$. Algorithm 3 does not store the graph $\overline{G}$. Whenever it is queried with a pair of vertices to check if it forms an edge, it recursively runs Algorithm 3 on all the connected components of G[V \S] separately (Lines 10 – 14 of Algorithm 2) and produces an answer. Finally, we perform same DFS like search as in Algorithm 1 on $\overline{G}$ (Lines 1 – 9 of Algorithm 2).

Algorithm 1. Algorithm ColoredDFS: One of the Building Blocks of RedBluePathDetect

Input : $G' = (V', E'), s', t', init, final$
Output : "Yes" if there is a red-blue path from s' and t' starts with *init* and ends with *final*
/* Use two sets- N_i, for i = 0,1, to store all the vertices that have been explored with the color value i */

1 **if** $s' \notin N_{init}$ **then**
2 Add s' in N_{init};
3 **for** *each edge* $(s', v) \in E'$ *of color value init* **do**
4 **if** $v = t'$ *and init = final* **then**

```
5          Return true;
6        end
7        Run ColoredDFS(G', v, t', init + 1(mod 2), final)
8      end
9  end
```

Algorithm 2. Algorithm ModifiedColoredDFS: One of the Building Blocks of RedBluePathDetect

```
Input    : Ḡ' = (V̄', Ē'), G', s', t', init, final
Output   : "Yes" if there is a red-blue path from s' and t' starts with init and ends with final
 /* Use two sets- R_i, for i = 0,1, to store all the vertices that have been explored with
    the color value i   */
1  if  s' ∉ R_init then
2      Add  s'  in R_init;
3      for each (s', v) ∈ (init,temp) Ē' for each temp ∈ {0,1} do
4          if v = t' and temp = final then
5              Return true;
6          end
7          Run ModifiedColoredDFS(Ḡ' , G', v, t', temp + 1 (mod 2), final)
8      end
9  end
  /*  "(u, v) ∈ (init,temp) Ē' ?" query will be solved using the following
      procedure     */

10 for every a ∈ V do
11      /*  V  be the set of vertices of G'   */
        /* V_a = the set of vertices of Ĥ's connected component containing a,
           where H = G[V  \ V̄']     */
12      if  RedBluePathDetect(G[V_a  ∪  V̄'], u, v, n, init, temp) is true then
13          Return true for the query;
14      end
15 end

15 Return false for the query;
   /* End of the query procedure    */
```

Algorithm 3. Algorithm RedBluePathDetect: Algorithm for Red-Blue Path in planar DAG

Input : $G', s', t', n, init, final$

Output : "Yes" if there is a red-blue path from s' and t' starts with *init* and ends with *final*

1 if $n' \leq n^{1/2}$ **then**

2 Run ColoredDFS($G', s', t', init, final$);

3 else

/* let $r' = n'^{(1-\epsilon)}$ */

4 Run PlanarSeparatorFamily on $\hat{G}'$ to compute r'-separator family $\overline{S}'$;

5 Run ModifiedColoredDFS($\overline{G}' = (\overline{S}' \cup \{s', t'\}, \overline{E}')$, $G', s', t', init, final$);

6 end

In the base case, we use Algorithm 1 which takes linear space and polynomial time. Thus due to the restriction of the size of the graph in the base case, we have $\tilde{O}(n^{1/2})$ space and polynomial time complexity. The sets N_0 and N_1 of algorithm ColoredDFS only store all the vertices of the input graph and we run ColoredDFS on a graph with $n^{1/2}$ vertices and it visits all the edges of the input graph at most once which results in the polynomial time requirement.

Proof of correctness: We now give a brief idea about the correctness of this algorithm. In the base case, we use similar technique as DFS just by alternatively exploring Red and Blue edges and thus this process gives us a path where two consecutive edges are of different colors. Otherwise, we also do a DFS like search by alternatively viewing Red and Blue edges and we do this search on the graph $H = (\overline{S}' \cup \{s, t\}, \overline{E}')$. By this process, we decide on presence of a path in H from s to t such that two consecutive edges are of different colors in G and the edge coming out from s is Red and the edge going in at t is Blue. This is enough as each path P in G must be broken down into the parts $P_1, P_2, \cdots, P_k$ and each P_i must be a sequence of edges that starts and ends at some vertices of $\overline{S}' \cup \{s, t\}$ and also alternates in color. We find each such P_i, just by considering each connected component of $G(V' \setminus \overline{S}')$ and repeating the same steps recursively.
Due to [6], we know that the reachability problem in directed graphs reduces to RedBluePath in planar DAGs.

Deciding Even Path in Planar DAGs

Given directed graph G and two vertices s and t, EvenPath is the problem of deciding the presence of a (simple) directed path from s to t, that contains even number of edges. We can view this problem as a relaxation of RedBluePath problem as a path starting with Red edge and ending

with Blue edge is always of even length. In this section, we establish a relation between EvenPath problem in planar DAG with detecting an odd length cycle in a directed planar graph with weight one (can also be viewed as an unweighted graph).

Lemma 1. *For directed planar graphs, for any constant $0 < \epsilon < 1/2$, there is an algorithm that solves the problem of deciding the presence of odd length cycle in polynomial time and $O(n^{1/2+\epsilon})$ space, where n is the number of vertices of the given graph.*

The above lemma is true due to the fact that we can do BFS efficiently for undirected planar graph and it is enough to detect odd length cycle in each of the strong components of the undirected version of the given directed planar graph. For undirected graph, presence of odd length cycle can be detected using BFS algorithm and then put red and blue colors on the vertices such that vertices in the consecutive levels get the opposite colors. After coloring of vertices if there exists an monochromatic edge (edge where both vertices get the same color), then we can conclude that there is an odd length cycle in the graph otherwise there is no odd length cycle. But this is not the case for general directed graph. However, the following proposition will help us to detect odd length cycle in directed graph.

In the following proposition, we use $u \rightarrow v$ to denote a directed edge (u, v) and $x \xrightarrow{P} y$ to denote a directed path P from a vertex x to y.

Proposition 1. *A strongly connected directed graph contains an odd length cycle if and only if the underlying undirected graph contains an odd length cycle.*

Proof. The forward direction follows trivially. Now to prove the reverse direction, we will use the induction arguments on the length of the odd cycle in the undirected version of the graph. The base case is when the undirected version of the graph contains a 3-length cycle. If the undirected edges present in the undirected cycle also form directed cycle when we consider the corresponding edges in the directed graph, then there is nothing to prove. But if this is not the case, then the Fig. 2 will depict the possible scenarios. As the graph is strongly connected, so there must be a path P from t to s and if this path does not pass through the vertex x, then any one of the following two cycles $s \rightarrow t \xrightarrow{P} s$ or $s \rightarrow x \rightarrow t \xrightarrow{P} s$ must be of odd length. Now suppose P contains the vertex x and thus $P = P_1P_2$, where P_1 is the path from t to x and P_2 is the path from x to s. It is easy to see that all the three cycles $s \rightarrow t \xrightarrow{P} s$, $x \rightarrow t \xrightarrow{P_1} x$ and $s \rightarrow x \xrightarrow{P_2} s$ cannot be of even length.

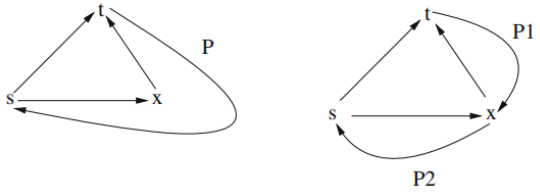


Fig. 2. For undirected cycle of length 3

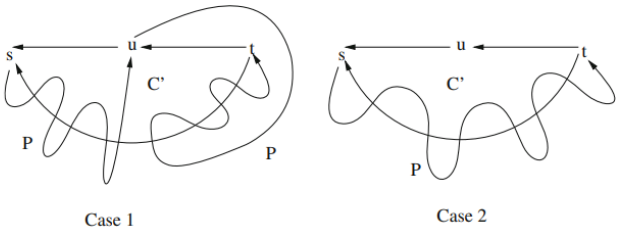


Fig. 3. For undirected cycle of length (k+2)

Now by induction hypothesis, assume that if the undirected version has a cycle of k-length (k odd), then there exists an odd length cycle in the original directed graph.
Now lets prove this induction hypothesis for any undirected cycle of length (k + 2). Consider the corresponding edges in the directed graph and without loss of generality assume that this is not a directed cycle. As (k + 2) is odd, so there must be one position at which two consecutive edges are in the same direction. Now contract these two edges in both directed and undirected version of the graph and consider the resulting k-length cycle in the undirected graph. So according to the induction hypothesis, there must be one odd length cycle C in the resulting directed graph. Now if C does not contain the vertex u (where we contract the two edges), then expanding the contracted edges will not destroy that cycle and we get our desired odd length cycle in the directed version of the graph. But if this is not the case, then consider C after expanding those two contracted edges ($t \rightarrow u \rightarrow s$), say the resulting portion is C'. If C' is a cycle, then there is nothing more to do. But if not, then consider the path P from s to t (there must be such path as the graph is strongly connected). Now there will be two possible cases: either P contains u or not. It is easy to see that for both the possible cases (case 1 and case 2 of Fig. 3 and in that figure every crossing of two paths denotes a vertex), all cycles generated by C' and P cannot be of even length. In case 1, if all the cycles generated by the paths $s \xrightarrow{P} u$ and $t \xrightarrow{C'} s$ and all the cycles generated by the paths $u \xrightarrow{P} t$ and $t \xrightarrow{C'} s$ are of even length, then as $t \xrightarrow{C'} s$ is of odd length, so the path $s \xrightarrow{P} u \xrightarrow{P} t$ must be of odd length. And then one of the following two cycles $s \xrightarrow{P} u \rightarrow s$ and $u \xrightarrow{P} t \rightarrow u$ is of odd length. Similarly in case 2, if all the cycles generated by $s \xrightarrow{P} t$ and $t \xrightarrow{C'} s$ are of odd length, then the path $s \xrightarrow{P} t$ is of odd length and so the cycle $s \xrightarrow{P} t \rightarrow u \rightarrow s$ is of odd length.

***Proof* (*of Lemma* 1)**. In a directed planar graph, any cycle cannot be part of two different strong component, so checking presence of odd cycle is same as checking presence of odd cycle in each of its strong components. Constructing strong components of a directed planar graph can be done by polynomial many times execution of DirectedPlanarReach algorithm (See Theorem 4), as a strong component will contain vertices x, y if and only if DirectedPlanarReach (G, x, y, n) and DirectedPlanarReach (G, y, x, n) both return "yes". And thus strong component construction step will take $\tilde{O}(n^{1/2+\epsilon})$ space and polynomial time. After constructing strong components, it is enough to check presence of odd cycle in the underlying undirected graph (according to Proposition 1). So now on, without loss of generality, we can assume that the given graph G is strongly connected. Now execute OddCycleUndirectedPlanar ($\hat{G}$, s, n) (Algorithm 4) after setting the color of s (any arbitrary vertex) to red. Here we adopt the well known technique used to find the presence of odd length cycle in a graph using BFS and coloring of vertices. In Algorithm 4, instead of storing color values for all the vertices, we only stores color values for the vertices present in the separator (Line 11) and we do the coloring recursively by considering the smaller connected components (Line 10). The algorithm is formally defined in Algorithm 4.

By doing the similar type of analysis as that of RedBluePathDetect, it can be shown that OddCycleUndirectedPlanar will take $O(n^{1/2+\epsilon})$ space and polynomial time and so over all space

complexity of detecting odd length cycle in directed planar graph is $O(n^{1/2+\epsilon})$ and time complexity is polynomial in n.

Now we argue on the correctness of the algorithm OddCycleUndirectedPlanar. This algorithm will return "yes" in two cases. First case when there is a odd length cycle completely inside a small region ($n' \leq n^{1/2}$) and so there is nothing to prove for this case as it is an well known application of BFS algorithm. Now in the second case, a vertex v in the separator family will get two conflicting colors means that there exists at least one vertex u in the separator family such that there are two vertex disjoint odd as well as even length path from u to v and as a result, both of these paths together will form an odd length cycle.

Now we are ready to prove the main theorem of this subsection.

Algorithm 4. Algorithm OddCycleUndirectedPlanar: Checking Presence of Odd Cycle in an Undirected Planar Graph

Input : $G' = (V', E')$, s', n, where G' is an undirected graph
Output : "Yes" if there is an odd length cycle

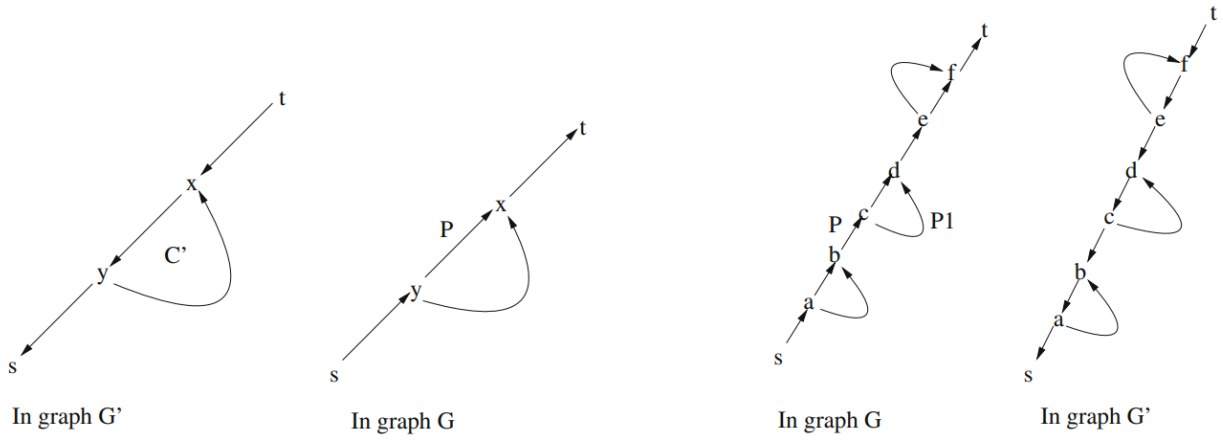
if $n' \leq n^{1/2}$ **then**
  Run BFS(G', s') and color the vertices with red and blue such that vertices in the alternate layer get the different color starting with a vertex that is already colored;
  if *there is a conflict between stored color of a vertex and the new color of that vertex or there is an edge between same colored vertices* **then**
    return "yes";
  end
else
  /* let $r' = n'^{(1-\epsilon)}$ */
  Run PlanarSeparatorFamily on $\hat{G}'$ to compute r'-separator family $\overline{S}'$;
  Set $S' := \overline{S}' \cup \{s'\}$;
  for *every* $x \in V'$ **do**
    /* V_x = the set of vertices of $\hat{H}$'s connected component containing x, where $H = G[V' \setminus S']$ */
    Run OddCycleUndirectedPlanar($G[V_x \cup S']$, s', n);
    Store color of the vertices of S' in an array of size $|S'|$;
  end
end

***Proof* (*of Theorem* 2)**. Given a planar DAG G and two vertices s and t, first report a path from s to t, say P, which can easily be done by polynomially many invocation of the algorithm DirectedPlanarReach mentioned in Theorem 4 and thus requires polynomial time and $O(n^{1/2+\epsilon})$

space. If the path P is not of even length, then construct a directed graph G' which has the same vertices and edges as G except the edges in path P, instead we do the following: if there is an edge (u, v) in P, then we add an edge (v, u) in G' . Now we can observe that the new graph G' is a directed planar graph.

Claim. G has an even length path if and only if G' has an odd length cycle.

Proof. Suppose G' has an odd length cycle, then that cycle must contains the reverse edges of P in G. Denote the reverse of the path P by P_{rev}. Now lets assume that the odd cycle C' contains a portion of P_{rev} (See Fig. 4). Assume that the cycle C' enters into P_{rev} at x (can be t) and leaves P_{rev} at y (can be s). Then in the original graph G, the path s P→ y C'→ x P→ t is of even length. Now for the converse, lets assume that there exists an even length path P_1 from s to t in G. Both the paths P and P_1 may or may not share some edges and without loss of generality we can assume that they share some edges (See Fig. 5). Now if we consider all the cycles formed by P_{rev} and portions of P_1 in G' , then it is easy to see that all the cycles cannot be of even length until length of P and P_1 both are of same parity (either both odd or both even), but this is not the case.



Now we can check the presence of odd length cycle in the graph G' in polynomial time and $O(n^{1/2+\epsilon})$ space (by Lemma 1).

References

- An O(n^(1/2+ϵ))-Space and Polynomial-Time Algorithm for Directed Planar Reachability
- A Sublinear Space, Polynomial Time Algorithm for Directed s-t Connectivity
- Simultaneous Time-Space Upper Bounds for Red-Blue Path Problem in Planar DAGs