Space Upper Bounds for Directed Graph Connectivity

Huanran Li

May 2, 2020

Abstract

We survey the following algorithms for improving the space upper bound of STCON problem. (1) A polynomial-time, $n/2^{\Theta(\sqrt{\log n})}$-space algorithm for directed graphs from Barnes, Buss, Ruzzo, and Schieber. (2) A polynomial-time $O(\log^2 n/\log\log n)$-space algorithm for Reach-Unambiguous graphs from Allender and Lange. (3) A polynomial-time $\widetilde{O}(n^{\varepsilon})$-space algorithm for Unique-Path graphs from Kannan, Khanna, and Roy. (4) A polynomial-time $\widetilde{O}(\sqrt{n})$-space algorithm for Planar graphs from Asano, Kirkpatrick, Nakagawa, and Watanabe. (5) A log-space reduction on Surface-embedded graphs with genus g from Stolee and Vinodchandran.

Contents

1 Introduction

In this survey, we focus on *directed st-connectivity* (STCON) problem, where given a directed graph G with two vertices s and t, we want to find if there is a path started from s and ended at t. The STCON problem can be solved with standard algorithms such as Depth First Search (DFS) and Breadth First Search (BFS) in polynomial time. For a graph with n vertices and m edges, these algorithms will take $O(m+n)$ time and $O(n \log n)$ space. Improving the space complexity upper bound has been researched for decades. In this survey, we are going to summarize several solutions on either directed graphs or subsets of directed graphs.

First, we are going to introduce some terms necessary in the preliminary section. In section 3, we introduce Short Path Recursive (SPR) algorithms, which marks the reachable nodes within range L^r in a given destination set that contains $\lceil n/k \rceil$ nodes in $O(k^{rL} L^r * n^2/k^2)$ time and $O(r(n/k + L \log k))$ space. Then, a combination of SPR and BFs will be introduced such that STCON can be solved in $O(k^{rL} L^r * n^2/k^2)$ time and $O(r(n/k + L \log k))$ space [BBRS98]. Based on the original paper, we also made a few modifications in the Combination of BFs and SPR to achieve the final time upper bound.

In sections 4 and 5, we introduce two algorithms that are primarily designed for Reach-Unambiguous graphs and Unique-path graphs. For Reach-Unambiguous graphs, Allender and Lange's algorithm, Graph Remap, aims to shrink the graph recursively in polynomial time and $O(\log^2 n / \log \log n)$ space [AL98]. For Unique-Path graphs, STCON can be solved by Kannan's D-Reach algorithm in $n^{O(1/\varepsilon)}$ time with $O(n^\varepsilon/\varepsilon)$ space [KKR08]. This algorithm was reproduced based on his base algorithm in $O(\sqrt{n})$ space and word description in the original paper.

In sections 6 and 7, we investigate algorithms for planar graphs and surface-embedded graphs. For planar graphs, we combined the results from Asano et al.'s paper [AKNW14] and Imai et al.'s paper [INP$^+$13] to solve the problem. The algorithms will recursively find the appropriate edge that separates the graph into two components and seek the connectivity from the edge to the sub-area. For the surface-embedded graphs with g genus, the algorithm from Stolee and Vinodchadran performs a reduction to create another directed subgraph that has $O(m+g)$ vertices in log-space [SV10].

2 Preliminaries

Given a directed graph G, we will use $V(G)$ and $E(G)$ to denote the set of vertices and edges in G. If G has n nodes, we assume V to be $\{v_1, v_2, ..., v_n\}$. For any edge $e \in E(G)$ that has a direction from x to y, we call x the *tail* denoted by $Tail(e)$, and y the *head* denoted by $Head(e)$.

For each pair of nodes (x, y), let $d(x, y)$ be the length of the shortest path between x and y. If x and y are not connected, $d(x, y)$ is infinite; $d(x, x)$ is 0. The two successors of an inner node x are denoted by $L(x)$ and $R(x)$. We let $T(x)$ denote the tree rooted at x.

Definition 1 ([AL98]). *G is called unambiguous if there is at most one path from s to t. G is called strongly unambiguous, or a $Mangrove$, if for any pair (x, y) of nodes there is at most one path leading from x to y. G is called reach-unambiguous if there is at most one path from s to any vertex $v \in V(G)$.*

Definition 2 ([KKR08]). *A path where no intermediate vertex is repeated is called a simple path. G is called unique-path with respect to a source vertex s if there is at most one simple path from s to any vertex $v \in V(G)$.*

Definition 3 ([AKNW14] [SV10])**.** *G is called planar if it can be drawn on a plane so that the edges intersect only at end vertices. G is called 2-cell embeddings if it is embedded on a surface S where every face is homeomorphic to an open disk.*

3 Algorithms on Directed Graphs

In this section, we are going to introduce two algorithms that both come from [BBRS98]. The first algorithm, Recursive Short Path (SPR), introduces a space-efficient way to find all reachable nodes in a given set using the recursive method. The second algorithm Combines SPR and BFs to achieve a better log-space upper bound with polynomial time.

3.1 Recursive Short Path (SPR)

This algorithm acts as a helper function for the major algorithm that will be introduced later. It takes two sets d_s and d_t, each of which contain at most n/k nodes. V_s will be a $\lceil n/k \rceil$-bit vector that marks any source node for the search in d_s. By recursively calling itself with r decreased by 1, SPR could return a $\lceil n/k \rceil$-bit vector that marks any reachable node in d_t within L^r distance from source nodes.

Algorithm 1: Short Path Recursive(SPR)

```
Input: $k$: number of sets, $n \geq k \geq 1$
       $L$: number of iterations, $L \geq 1$
       $r$: searching range, $r \geq 1$, $L^r \leq n$
       $d_s$: list of $\lceil n/k \rceil$ nodes that contains source nodes
       $d_t$: list of $\lceil n/k \rceil$ nodes where any reachable node contained will be indicated by
       the return set $V_t$
       $V_s$: $\lceil n/k \rceil$-bit vector that marks source nodes contained in $d_s$
Output: $V_t$: $\lceil n/k \rceil$-bit vector that marks any reachable node in $d_t$
1  Creat $V_0$, $V_1$, and $V_t$ three $\lceil n/k \rceil$-bit vectors. Set all bits in $V_t$ to zero.
2  if $r = 0$ then
3      forall $u$ in $d_s$ marked in $V_s$ and $v$ in $d_t$ do
4          if $(u, v)$ is an edge then mark $v$ in $V_t$ ;
5  else
6      forall possible sequence $\langle d_0 = d_s, d_1, ..., d_{L-1}, d_L = d_t \rangle$ do
7          $V_0 \leftarrow V_s$
8          for $i = 1$ to $L$ do
9              $V_{i \bmod 2} \leftarrow SPR(k, L, r-1, d_{i-1}, d_i, V_{(i-1) \bmod 2})$
10         Set all bits in $V_t$ that are set in $V_{L \bmod 2}$
11 return ($V_t$)
```

Theorem 1 ([BBRS98])**.** *For arbitrary integers r, k, and L, such that $r \geq 1, L \geq 1, n \geq 1$, and $L^r \leq n$, the recursive short paths algorithm SPR can search to distance L^r in time $O(k^{rL} L^r * n^2/k^2)$ and space $O(r(n/k + L \log k))$*

Proof of Correctness. Base Case: If $r = 0$, line 2-6 will be executed, which marks any node that is connected by the nodes marked in V_s. Therefore, the SPR will output the correct result.

Inductive Hypothesis: For $0 \leq r \leq k$, assume $SPR(k, L, r, d_s, d_t, V_s)$ will output V_t with marked nodes that are at most L^r distance away from the nodes marked in V_s.

Inductive Step: For $r = k + 1$, $SPR(k, L, k, d_s, d_t, V_s)$ will be called L times for each possible combination. Each calling time will result in saving the nodes that are at most L^r distance further away from the nodes in V_s. Finally, after L times, the nodes marked in $V_{L \bmod 2}$ will be at most $L * L^r$ distance from V_s. Therefore, the SPR will return the correct result with the nodes that are at most L^{r+1} distance away from V_s □

Proof of Space Complexity. We know that each set d_i will have at most k nodes. For saving one single sequence of $\langle d_0 = d_s, d_1, ..., d_{L-1}, d_L = d_t \rangle$, we need $O(L \log k)$ space. For saving V_0, V_1, and V_t, we need $O(n/k)$ space. So, for a single recursive call, there will be $O(n/k + L \log k)$ local space required. Since r is decreasing by 1 each time, there will be at most r calls holding in the stack. Therefore, the total space upper bound will be $O(r(n/k + L \log k))$. □

Proof of Time Complexity. For the base case, it will cost $O((n/k)^2)$ time for checking any pair (u, v) for $u \in d_s$ and $v \in d_t$, since d_s and d_t both have at most n/k nodes.

For the inductive step, there will be k^L different combinations of test sequences and each sequence calls $SPR(r-1)$ L times . Besides, it costs $O(n/k)$ times to execute line 10. Therefore, the time complexity could be represented as following equation:

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

By solving this equation, we find the time upper bound as

$$O(k^{rL} L^r * n^2/k^2)$$

. □

3.2 Combination of BFs and SPR

Algorithm 2 shown below is the main algorithm for STCON. The idea of this algorithm is to divide the whole graph into n/L^r classes based on each node's shortest distance to s. For a group of nodes that have the same shortest distances to s, we will refer to the group as a level. After traversing each class, a partial set is constructed with all reachable nodes within n/L^r levels that have exactly L^r distances in between. The first level will have j distance from node s. We set $j \in [0, L^r - 1]$, and therefore, there will be L^r different partial sets that could be constructed. The partial set will be abandoned if its size is over n/L^r. Finally, if we find any partial set that has the less than n/L^r nodes, we are going to search if t is within L^r distance of all nodes in the founded partial set.

The algorithm consists of two major for-loops, which are lines 3-6 and lines 7-23. For the simplicity of the following proof, we call it the first and second major for-loop. The first major loop adds all vertices that have the shortest distance j from s. The second major loop will add reachable nodes in one level to the partial set each time. The "if" statements in line 5 and 21 are where the space usage is directly constrained. Both perform the same function of skipping the current j when the size of the partial set is off the bounds n/L^r.

Algorithm 2: Combination of BFs and SPR [BBRS98]

```
Input: $s$: source node
       $t$: termination node
       $G$: the graph with $n$ nodes
Output: Connectivity: Connected/Not Connected
for $j = 0$ to $L^r - 1$ do
    $S \leftarrow \{s\}$
    /* First major for-loop */
    forall vertices,$v$ do
        if $d(s, v) = j$ then
            if $|S| > n/L^r$ then try next $j$ ;
            else add $v$ to $S$;
    /* Second major for-loop */
    for $i = 1$ to $\lfloor n/L^r \rfloor$ do
        $S' \leftarrow \emptyset$
        for $i_1 = 0$ to $k - 1$ do
            $S_{i_1} \leftarrow$ {all vertices whose vertex number mod $k = i_1$}
            $P \leftarrow \emptyset$
            for $i_2 = 0$ to $(k - 1)/L^r$ do
                $S_{i_2} \leftarrow i_2^{th}$ block in $S$
                $Q \leftarrow \{1^{|S_{i_2}|}\}$
                if $|S_{i_2}| < \lceil n/k \rceil$ then
                    Insert $(\lceil n/k \rceil - |S_{i_2}|)$ of '0's in the back of $Q$
                    Insert $(\lceil n/k \rceil - |S_{i_2}|)$ nodes that are not in $S$ in the back of $S_{i_2}$
                $A \leftarrow$ {all vertices in $S_{i_1}$ within distance $L^r$ of a vertex in $S_{i_2}$} // call
                    $SPR(k, L, r, S_{i_2}, S_{i_1}, Q)$
                $B \leftarrow$ {all vertices in $S_{i_1}$ within distance $L^r - 1$ of a vertex in $S_{i_2}$}
                $P \leftarrow P \cup (A - B)$
            if $|S| + |S' \cup P| > n/L^r$ then try next $j$;
            else $S' = S' \cup P$;
        $S \leftarrow S \cup S'$
    if $t$ within distance $L^r$ of a vertex in $S$ then return Connected;
    else return Not Connected;
```

Theorem 2 ([BBRS98])**.** *The combined algorithm of modified BFs and Recursive Short Path solves STCON in space $O((n \log n)/L^r + r(n/k + L \log k))$ and time $O(n^3 k^{rL})$ for any integer r, k, and L that satisfy $n \geq k \geq 1$, $r \geq 1$, $L \geq 1$, and $L^r \leq n$.*

Proof of Correctness. In the following proof, we will prove the correctness separately on whether a partial set is successfully built or not (i.e. line 5 and 21 never return a True when a partial set is successfully built).

Claim 2.1. *If the "if" statement returns false every time at line 5 and 21, theorem 1 holds.*

Proof of Claim 2.1. If the "if" statement at line 5 always returns false, all vertices that have distances j to s will be added into S.

Lines 18 - 20 will save all nodes in S_{i_1} that have the shortest distance L^r from any nodes in S_{i_2}, which is a subarray of S. If the "if" statement at line 21 always returns false, two inner for-loops at line 9 and line 12 will iterate all possible combinations of S_{i_1} and S_{i_2} from k sets in the graph. Hence, all nodes in the graph will be searched and at every iteration, only the nodes that have distance L^r from any saved nodes in the partial set will be added into the set.

Since we know $j \in [0, L^r]$, we can conclude for any pair of nodes (u, v) where $u \in S$ and $v \in G$, $d(u, v) \leq L^r$. Therefore, lines 24-25 will successfully determine if t is reachable from s. □

Claim 2.2. *If the "if" statement returns a True at either line 5 and 21, there must be another partial set that has a size less than or equal to n/L^r.*

Proof of Claim 2.2. We know that there will be L^r different sets that do not intersect with each other. If we assume that all sets have sizes larger than n/L^r, the total size of G will be greater than n, which contradicts with the fact that $n = |G|$. □

Based on Claim 2.1 and 2.2, we can conclude that regardless of whether the current partial set is built or abandoned, there will be at least one partial set that satisfies $|S| \leq n/L^r$. The reachability of t will always be determined based on this partial set. □

Proof of Space Complexity. We need $\log n$ bits for each node. To save the whole partial set, we need $O((n \log n)/L^r)$ space. In addition to the space usage from SPR, which is $O(r(n/k + L \log k))$, we have $O((n \log n)/L^r + r(n/k + L \log k))$ for the space upper bound. □

Proof of Time Complexity. We know that every call of SPR will take $O(k^{rL} L^r * n^2/k^2)$ time. There will be at most $k^2 n/L^r$ calling time, which results in $O(k^{rL} L^r * n^2/k^2 * k^2 n/L^r) = O(n^3 k^{rL})$ for the time upper bound. □

4 Algorithm on Reach-Unambiguous Graphs

Graph Remap was first introduced in [AL98]. It takes three inputs, which are node s, node t, and integer r. Node s represents the root of the tree that we are going to shrink. Node t is the destination node, and integer r is the maximum depth of the current tree that has been searched. In the algorithm, we denote $T_r(z) = \{y \in V | d(z, y) \leq r\}$ (i.e. the subtree that only contains the nodes that are within a distance r from node z). $LCA(M)$ represents the least common ancestor of all the nodes in the set M.

We denote $f(s)$ to be the "special root" from the tree rooted by s. The "special root" could be s itself when all nodes in $T(s)$ can be reviewed. Otherwise, the "special root" will be a node z where $z \in T(s)$ such that $T(z)$ is the smallest tree whose nodes cannot be reviewed. We also marked s with $\varphi'(s)$ based on the result of the search. Depending on whether t is found or not in $T(s)$, $\varphi'(s)$ will be marked with *Connected/Not Connected.* If the search cannot be completed, $\varphi'(s)$ will be marked with *Unknown.*

Finally, based on Algorithm 3, we construct a new graph G' by setting $L'(x) \leftarrow f(L(f(x)))$ and $R'(x) \leftarrow f(R(f(x)))$ when $\varphi(x)' = Unknown$. The construction starts from $x = s$ and recursively keep finding children nodes for the added nodes.

Algorithm 3: Graph Remap

```
Input: $s$: source node
       $t$: termination node
       $r$: searching range, $1 \le r \le \log n$
       $T(s)$: Reach-Unambiguous graph rooted at $s$
Output: $f(s)$: the special node of $s$
        $\varphi'(s)$: Connectivity of $s$
1  $curr \leftarrow s$
2  $M \leftarrow \{y \in T_r(curr) | d(curr, y) = r, \varphi(y) = i\}$
3  while $Connected \notin \varphi(T_r(curr))$ and $M \neq \emptyset$ and $LCA(M) \neq curr$ do
4      $curr \leftarrow LCA(M)$
5      $M \leftarrow \{y \in T_r(curr) | d(curr, y) = r, \varphi(y) = Unknown\}$
6  if $+ \in \varphi(T_r(curr))$ then
7      $\varphi'(s) \leftarrow Connected$
8      $f(s) \leftarrow s$
9  else if $M = \emptyset$ then
10     $\varphi'(s) \leftarrow Not\ Connected$
11     $f(s) \leftarrow s$
12 else
13     $\varphi'(s) \leftarrow Unknown$
14     $f(s) \leftarrow curr$
```

Theorem 3 ([AL98]). *$RUSPACE(\log n) \subseteq DSPACE(\log^2 n/\log\log n)$.*

Proof. To solve the connectivity problem on the Reach-Unambiguous graph in $O(\log^2 n/\log\log n)$ space, Graph Remap needs to be called multiple times recursively. We will first prove the space saved by constructing a shrunk graph once. Based on that, the final space upper bound can be constructed by calculating how many recursion processes will exist at the same time.

Claim 3.1. *For each $x \in V$, and $\varphi'(x) = Unknown$, we have*

$$|T(L(f(x)))| \ge 2r + 1$$

Proof of Claim 3.1. Since $\varphi'(x) = Unknown$, we know that there is a set $M = \{y \in T_r(z) | d(z, y) = r, \varphi(y) = Unknown\}$ and $M \neq \emptyset$. Moreover, $f(x) = z = LCA(M)$.

Given that $M \neq \emptyset$, it can be proved that $T(z)$ has r depths excluding the leaf nodes. Therefore, $T(L(z))$ will have r depths including the leaf nodes. Since G will be a complete tree, the claim 3.1 holds and the same holds for the right leaves. □

Claim 3.2.

$$|G'| \le |G|/r$$

Proof of Claim 3.2. From the rule of G' construction and Claim 3.1, we are given that when $\varphi(x)' = Unknown$,

$$L'(x) = f(L(f(x)))$$

$$|T(L(f(x)))| \ge 2r + 1$$

Assume that $L'(x)$ is a leaf in G', then $L'(x)$ in G' will represent at least $2r + 1$ nodes in G, which means that for every leaf in G', we ignored at least $2r$ nodes from G.

Given that at least half of the nodes in the complete tree G' will be leaf nodes, we have

$$|G'|/2 * 2r \leq |G|$$

which can be simplified as

$$|G'| \leq |G|/r$$

□

Based on Claim 3.2, we can determine the connectivity by checking $\varphi(s) = \mathit{Connected}/\mathit{Not\ Connected}$ when we have $|G'| = 1$. This can be done by repeating the process of shrinking for $O(\log_r n)$ times. By setting $r = \log n$, we have $O(\log n / \log\log n)$.

Given that each shrinking process will take $r + O(\log n)$ space, the final space upper bound will be $O(\log^2 n / \log\log n)$. □

5 Algorithm on Unique-Path Graphs

This section will introduce three algorithms, which are *D-Reach* and its helper functions *backtrack* and *discovery*. All algorithms are from the paper [KKR08]. The main algorithm *D-Reach* is mostly based on DFS. By continuously exploring new nodes and backtracking, the algorithm can have less space usage while maintaining a polynomial time complexity. However, to achieve a $O(\frac{n^\varepsilon}{\varepsilon})$ space upper bound, DFS is not space-efficient enough.

The improvement Kannan and his fellows have is by keeping a set of landmarks instead of keeping a stack for the entire searching path. The landmarks are vertices that have exactly r/n^ε distance between them so that all landmarks can be stored within space $O(n^\varepsilon)$. However, this improvement will also introduce two other problems: how to backtrack and how to determine whether a new reachable node has already discovered on the path. These problems can be eliminated by introducing a recursive procedure, which will be explained in algorithms later.

Algorithm 4: backtrack($curr, r$)

```
Input: curr: current node
       r: search range
Output: prev: the parent of the current node along the path from s to curr
1 forall vertex $v \in Parents(curr)$ do
2     $s \leftarrow$ last element in landmarks
3     $result \leftarrow$ D-Reach($s, \{curr\}, G - (v, curr), r$)
4     if $result = Not\ Connected$ then
5         return $v$
6 return
```

Algorithm 5: D-Reach(s,{t},G, r)

```
Input: $s$: Source Node
       $\{t\}$: a set of Termination Nodes
       $G$: Unique-Path graph
       $r$: searching range
Output: $result$ : $Connected/Not\ Connected$
1  $n \leftarrow |G|$; $curr \leftarrow s$; $child \leftarrow NULL$; $result \leftarrow Unknown$;
2  $distance \leftarrow 0$; $landmarks \leftarrow \{s\}$; $TotalDistance \leftarrow 0$;
3  if $d \leq n^{\varepsilon}$ then
4      Perform a d-Bounded DFS
5      return $result$
6  while $result = Unknown$ do
7      if $child$ is the last vertex in $Successors(curr)$ or $TotalDistance > r$ then
8          if $curr = s$ then  $result = Not\ Connected$ ;
9          $child \leftarrow curr$
10         $distance \leftarrow distance - 1$
11         $TotalDistance \leftarrow TotalDistance - 1$
12         if $distance < 0$ then
13             Remove $curr$(last element) to $landmarks$
14             $distance \leftarrow \lceil r/n^{\varepsilon} \rceil - 1$
15         $curr \leftarrow Backtrack(curr, r/n^{\varepsilon})$
16         if $curr$ then
17     else
18         if $child = NULL$ then
19             $next \leftarrow$ first vertex in $Successors(curr)$
20         else if $child$ is not the last vertex in $Successors(curr)$ then
21             $next \leftarrow$ next vertex after $child$ in $Successors(curr)$
22         if $discovery(curr, next) = Connected$ then
23             $child \leftarrow next$
24         else
25             if $next \in \{t\}$ then $result = Connected$;
26             $curr \leftarrow next$
27             $child \leftarrow NULL$
28             $distance \leftarrow distance + 1$
29             $TotalDistance \leftarrow TotalDistance + 1$
30             if $distance = \lceil r/n^{\varepsilon} \rceil$ then
31                 Append $curr$ to $landmarks$
32                 $distance \leftarrow 0$
33 return $result$
```

```
Algorithm 6: discovery($curr, next, r$)
  Input: $curr$: current node
         $next$: the node that needs to be checked to see if it is on the path from $s$ to $curr$
         $r$: search range
  Output: $True/False$
  /* Instead of returning Connected or Not Connected, D-Reach-All will find
     all connected nodes and return a list that contains them.            */
1 $M \leftarrow$ D-Reach-All($next, landmarks \cup \{curr\}, G, r$)
  /* Assume $M$ is sorted with the respect to the distance from $s$ with the
     closest node being in the first place                                */
2 $p \leftarrow |M|$
3 if $p = 0$ then return $False$ ;
4 else
5     $z \leftarrow M[p-1]$ // furthest node from $s$ in $M$
6     if $z = s$ then return $False$;
7     else if $z = curr$ then
8         return D-Reach($M[p-3]$, $\{next\}$,$G, 2r$)
9     else
10        return D-Reach($M[p-2]$, $\{next\}$,$G, r$)
```

Theorem 4 ([KKR08]). *For any $\varepsilon \in (0, 1]$, STCON is solvable in $n^{O(\frac{1}{\varepsilon})}$ time with $O(\frac{n^\varepsilon}{\varepsilon})$ space in unique-path graphs.*

Proof of Time Complexity. We denote $T(m, n, d)$ to be the time upper bound for the D-Reach with unique-path graph G, which has n nodes and m edges. d will represent the largest distance the algorithm is allowed to search. For the original call, $d = n - 1$. Note that D-Reach-All introduced in *discovery* should have the same time upper bound as D-Reach.

For each recursive call, we know that there will be at most m calls of *backtrack* and *discovery*. Each time *backtrack* is called, D-Reach will be executed multiple times. We don't know how many times D-Reach will be executed within each *backtrack*, but the total time will be m. This is because every time, a unique edge will be deleted before calling D-Reach. For the *discovery*, there will be 1 D-Reach-All and 1 D-Reach with either r or $2r$.

Therefore, we have the following formula:

$$T(m, n, n) \leq 2mT(m, n, n^{1-\varepsilon}) + m \max(T(m, n, n^{1-\varepsilon}, T(m, n, 2n^{1-\varepsilon})) + O(m + n)$$

From the algorithm, we also know the base case time upper bound:

$$T(m, n, n^\varepsilon) = O(n + mn^\varepsilon) = (n + m)^{O(1)}$$

Based on those two formulas, we have:

$$T(m, n, n) \leq (n + m)^{O(1+\frac{1}{\varepsilon})}$$

$$T(m, n, n) \leq n^{O(\frac{1}{\varepsilon})}$$

□

Proof of Space Complexity. At each recursion call, we need $O(n^{\varepsilon})$ space for saving the *landmarks* and performing a bounded DFS if necessary. We know that every time we call the recursion inside D-Reach, we divide the search range r by n^{ε}. With initial $r = n$, we have $depth \leq \frac{1}{\varepsilon}$. Therefore, the final space upper bound will be $O(\frac{n^{\varepsilon}}{\varepsilon})$. □

6 Algorithm on Planar Graphs

The main idea of the algorithm is to recursively find the connection between separators, to label the reachable nodes at every round. The separator is formed with a list of cycles, which divide the graph into two sub-graphs. The procedure to generate those separators(S) is called *CycleSep*, which requires the input graph to be triangulated. Therefore, an additional list(T) of edges has to be added to the graph by *AddTri* in order to find the separator. First, we explain the necessary terminology and background that strongly follows [INP$^+$13] and [AKNW14]. Then, we introduce the algorithms *CycleSep* and *ExtendReach*.

6.1 Preliminaries

We first introduce the terms and backgrounds for the algorithm *CycleSep*. Given a planar and triangulated graph G, we define the *face-vertex* graph $G' = (V', E')$, where V' is a set of triangular faces, and E' is a set of face pairs that share the same vertex in G. To distinguish from the regular graph G, we denote *tr-vertex* $\in V'$ and *tr-edge* $\in E'$. Assuming there are two *tr-vertices* f_1 and f_2, we let *tr-path*(f_1, f_2) denote a sequence of *tr-vertices* from f_1 to f_2 such that for every adjacent pair of *tr-vertices* in the sequence, there is a *tr-edge* between them. Accordingly, we have *tr-dist*$(f_1, f_2) = \min |$ *tr-path*$(f_1, f_2)|$.

If two *tr-vertices* share the same edge $e \in E$, we call them *edge-connected*. A region R is a list of *tr-vertices* such that for any two *tr-vertices* $u, v \in$ *region*, (i) there is a sequence $\langle f_1, f_2, ..., f_i\rangle$ where (u, f_1), (f_{j-1}, f_j), and (f_i, v) are *edge-connected* for $j \in [2, i]$, (ii) $f_j \in R$ for $j \in [1, i]$. The boundary $B(R)$ of the region R is a set of edges that only lies on a single *tr-vertex* in R.

Definition 4. *For any tr-vertex $f \in V'$, the k-neighborhood of f (N_k) represents a set consisting of the closest k number of tr-vertices to f in G'. If there are ties at the farthest distance, the tr-vertex with a smaller index will be selected first.*

Definition 5. *A k-maximal independent set I is a set of tr-vertices such that (i) for every $f_1, f_2 \in I$, $N_k(f_1) \cap N_k(f_2) = \emptyset$, (ii) for every $f_1 \notin I$, there exists a $f_2 \in I$ such that $N_k(f_1) \cap N_k(f_2) \neq \emptyset$.*

From the above definition, we know that the size of *k-maximal independent set* is $O(n/k)$. A straight forward greedy algorithm can be implemented to calculate the *k-maximal independent set*. Thereby, we have the following conclusion.

Lemma 5. *A k-maximal independent set can be calculated with input planar graph G in polynomial time and $\widetilde{O}(n/k + k)$ space.*

We denote a set of *tr-vertices* with $l(f, r)$ such that for all $v \in l(f, r)$, *tr-dist*$(f, v) = r$. For a given region R, we let *tr-dist*$(R, f) = min_{g \in R}\{$*tr-dist*$(g, f)\}$. Now we can represent the core and Voronoi region of f with the following definition.

Definition 6. *The core(f) is defined as a union of set $l(f, r)$ for $r \in [1, r_{max}]$, where $r_{max} = \max\{r | l(f, r) \subset N_k(f)$ and $|l(f, r)| \leq \sqrt{k}\ \}$.*

Definition 7. *For any $f \in$ k-maximal independent set I, we can determine if a tr-vertex g is in the Voronoi region of f by following two conditions: (i) $g \in N_k(f)$, (ii) $g \notin N_k(f_i)$ for any $f_i \in I$, and tr-dist(f_i, g) is the smallest at $f_i = f$.*

Note: For any *tr-vertex g*, if there is more than one f_i that satisfies condition (ii), then g belongs to each of their Voronoi regions. g will be called as a *Voronoi vertex* if g belongs to three or more Voronoi regions.

Next, two essential theorems will be introduced from [LT94] and [Mil96], which will be referred to in the algorithm *CycleSep*.

Theorem 6 ([LT79])**.** *Given a planar graph G that has non-negative vertex costs summing to no more than one, Lipton and Tarjan's algorithm can find a $2/3$-separator with its size no more than $2\sqrt{2}\sqrt{n}$. This algorithm can be implemented in polynomial time.*

Theorem 7 ([Mil84] [INP+13])**.** *If G is an embedded triangulated planar graph whose weights sum to 1 and no face has weight larger than $2/3$, then there exists either a vertex that is a weighted separator or a simple $2/3$-cycle separator of size at most $2\sqrt{2n}$. Further, the separator can be found in linear time and $O(d\sqrt{m})$ space, where d is the maximum face size, and m is the number of faces.*

Now we introduce preliminaries for the algorithm *ExtendReach*. The separator generated by *CycleSep* can have at most $c_{sep}\sqrt{n}$ vertices. Furthermore, two new subsets derived from this separator V^0, V^1 will have the following property for $b \in \{0, 1\}$:

$$|V^b| \leq \frac{2}{3}|V| + c_{sep}\sqrt{|V|}$$

By combining a separator with its additional list and a binary variable that indicates the side of the separator, we can represent a sub-graph surrounded by the separator. In detail, the form of this triple is (b, S, T) where $b \in [0, 1], S = \{C_1, C_2, ..., C_h\}$, T is a list of edges, and C contained by S is a list of nodes.

Since the algorithm will recursively find more separators, the sequence of separators is necessary for determining the new sub-graph. Thereby, if we denote the sub-graph generated by a separator as $G_{new} = [G]^b_{S,T}$, then after d recursions, the sub-graph will be:

$$G_{new} = \left[... \left[[G]^{b_1}_{S_1,T_1} \right]^{b_2}_{S_2,T_2} ... \right]^{b_d}_{S_d,T_d}$$

In order to construct G_{new}, we let M denote a sequence of triples as $M = \langle (b_1, S_1, T_1), (b_2, S_2, T_2), ..., (b_d, S_d, T_d) \rangle$. At every recursion, we may need to add a new separator into the sequence, and we will mark this action with $M \cup newTriples$. The sub-graph G_{new} will be represented as $[G]_M$ to indicate that it is constructed from the sequence M. Moreover, we use $G[U]$ to represent the sub-graph of G induced by U, where $U \subset V(G)$.

We will utilize the *universal sequence* σ_s to help with reducing searching time. With $\diamond$ representing sequence concatenation, it is defined as:

$$\sigma_s = \begin{cases} \langle 1 \rangle & s = 0 \\ \sigma_{s-1} \diamond \langle 2^s \rangle \diamond \sigma_{s-1} & s > 0 \end{cases}$$

6.2 Algorithm

Lemma 8. *Additional edges for the triangulated graph can be calculated by $AddTri$ in polynomial time and $O(\log n)$ space.*

Proof. From Allender and Mahajan [AM04], we know that the planarity test and graph embedding could be reduced to undirected graph reachability problem. Thereby, by the algorithm *UReach* introduced by Reingold [Rei08], a graph's planar embedding can be calculated in $O(\log n)$ space and polynomial time. With the planar embedding, it is not hard to find the additional edges that make the graph triangulated. In detail, for each face, we are adding edges between the smallest indexed vertex to all other vertices on the face. □

Theorem 9 ([INP+13])**.** *Given a planar undirected triangulated graph G, an $(8/9)$-separators can be calculated by CycleSep in $\widetilde{O}(\sqrt{n})$-space upper bound and polynomial time.*

Note: By applying the algorithm multiple times, the larger side of the (8/9)-separators can be called to split again, which finally will produce a (2/3)-separators.

Algorithm 7: CycleSep

```
Input: $G(V,E)$: Planar and Triangulated graph
Output: $result$: Cycle Separator
1  $I \leftarrow$ k-maximal independent set
2  forall $f \in I$ do
3      if $|V(f)| \geq n/3$ then
4          Apply the algorithm of Lipton and Tarjan[LT79] on BFS tree of $V(f)$ rooted at $f$
5          return result
6  Initialize empty subgraph $H(V_H, E_H)$
7  forall $f \in I$ do
8      add $B(core(f))$ to $H$
9  forall $v \in V$ do
10     if $v$ is a Voronoi vertex then
11         add $B(core(v))$ to $H$
12 forall $f, g \in V$ and $B(V(f)) \cap B(V(g)) \neq \emptyset$ do
13     $List \leftarrow$ all Voronoi vertices in $B(V(f)) \cap B(V(g))$
14     forall $v \in List$ do
15         $p \leftarrow$ a tr-path from $f$ to $v$
16         $p' \leftarrow$ part of $p$ that is outside cores of $f$ and $v$
17         Add $p'$ to $H$
18         Do the same between $g$ and $v$
19 forall face $h \in H$ do
20     $n_h \leftarrow$ the number of vertices of G that lies inside of $h$
21     $weight(h) \leftarrow n_h/n$
22 Apply Miller's algorithm[Mil86] on graph $H$
23 return result
```

Proof of Correctness. We divide the algorithm into 2 separate cases. The first case (line 2 - 6) focuses on finding any Voronoi region that has more than $n/3$ vertices. Assume there is one f

found such that $|V(f)| \geq n/3$, then a subgraph $G_{V(f)}$ induced by $V(f)$ will be passed into Lipton and Tarjan's algorithm. By assigning the cost of each vertex with $1/n$, we will have the graph $G_{V(f)}$ that has non-negative vertex costs summing to no more than one. Thus, by theorem 6, a (2/3)-separator can be calculated for graph $G_{V(f)}$. Here, we denote two sides separated to be V^0 and V^1, where $|V^0| \leq |V^1|$.

Since $|G_{V(f)}| \geq n/3$, if we consider the smaller side V^0, we know that

$$|V^0| \geq 1/3 * n/3 = n/9$$

Thereby, the separator will be an (8/9)-separator for input graph G for the first case.

For the second case (line 6 - line 22), the algorithm focuses on building a subgraph H that can be used in Miller's algorithm. From Theorem 7, the following conditions are required for its algorithm to output a 2/3-cycle separator: (i) weights sum to 1, (ii) every weight in H is no larger than 2/3.

From line 19 - 22, it is clear that the way we assigned weights will fulfill the first requirement. Since no $Voronoi\ region$ in G' has a size larger or equal to $n/3$, we know that the faces added by line 7 - 11 will not cover more than $2/3n$ vertices in G. For other faces that are added by line 12 - 19, they can not be a subset of more than two $Voronoi\ regions$, which means that the vertices that each of them covers can not be larger than $2/3n$. Based on the way we assign weights, we know that no weight will be larger than 2/3.

Therefore, Miller's algorithm will output the correct result. □

Proof of Space Complexity. First, we are going to prove that Lipton and Tarjan's algorithm will take a BFS tree with the diameter $O(k)$ in $\widetilde{O}(k)$-space and polynomial time. Then, we will show that the subgraph H has $O(n/k)$ faces and each face has $O(\sqrt{k})$ vertices.

Given $|N_k(f)| = k$, it takes $O(k)$ space to construct the BFS tree of $N_k(f)$. From Definition 7, we know that $N_k(f) \subseteq V(f)$. For any $g \in V(f)$ and $g \notin N_k(f)$, a new BFS tree can be constructed on $N_k(g)$. By finding the common *tr-vertex* $f' \in N_k(f) \cap N_k(g)$ with the smallest distance between f' and g, we can add the partial path to the BFS rooted by f. Thereby, the computation can be done in $\widetilde{O}(k)$ space and the BFS tree has $O(k)$ depth.

For each face in H, if it is added by line 7-11, by Definition 6, we know that its size cannot be larger than $\sqrt{k}$. For any *tr-vertex* $g \in N_k(f)$ and $g \notin core(f)$, there will be at least $\sqrt{k}$ other *tr-vertices* that have the same distance to f. Thereby, there always exists a *tr-vertex* f' such that $tr\text{-}dist(g, f') \leq \sqrt{k}$, which makes all the faces added by line 12-18 have a size $O(\sqrt{k})$.

For the number of faces, we know that the total number of faces added by line 7-11 will be bounded by $|I|$, which is $O(n/k)$. For the faces added by line 12-19, all of them consist of two cores and two Voronoi vertices, which makes the total amount bounded by $O(n/k)$ as well.

Therefore, by theorem 6 and 7, we could have both algorithms running in $O(\sqrt{n})$ space by setting $k = \sqrt{n}$. □

Algorithm 8: ExtendReach(M, r)

```
Input: $M$: A sequence of triples of a binary label, a cycle-separator, and an additional
       triangulation edge list
       $r$: search range
Global: $A$: A list of vertices, initialized to be $\{s, t\}$
        $R$: A list of booleans that represent whether the vertex in $A$ is reachable. It is
        initialized to be $\{True, False\}$
1  if $|[G]_M| < 144c_{sep}^2$ then
2      $R_t \leftarrow \{u \in A | R[u] = True\}$
3      forall vertex $v \in A$ do
4          Perform a BFS search in $G[A \cup V_M]$bounded with range $r$ from some $u \in R_t$ to $v$
5          $R[v] \leftarrow True$ if $v$ is reached
6  else
7      Use CycleSep and AddTri to create a new cycle separator $S_{t+1}$ of $[G]_M$ and its
        additional triangulation edge lists $T_{t+1}^0$ and $T_{t+1}^1$;
8      $S'_{t+1} \leftarrow S_{t+1} - A$
9      $A \leftarrow A \cup S'_{t+1}$
10     $R[v] = False$ for all $v \in S'_{t+1}$
11     forall $i \in [1, 2r - 1]$ do
12         $c_i \leftarrow \sigma_s[i]$
13         ExtendReach$(\langle M \cup (0, S_{t+1}, T_{t+1}^0)\rangle, c_i)$
14         ExtendReach$(\langle M \cup (1, S_{t+1}, T_{t+1}^1)\rangle, c_i)$
15     $A \leftarrow A - S'_{t+1}$
```

Theorem 10 ([AKNW14])**.** *For any input instance G, s, and t of the planar directed graph reachability problem, their planar graph reachability algorithm determines whether there is a path from s to t in G in $\widetilde{O}(\sqrt{n})$-space upper bound and polynomial-time.*

Proof of Correctness. The algorithm separates the situations into two cases based on $|[G]_M|$. For the first case, which is when $|[G]_M| < 144c_{sep}^2$, the algorithm is straightforward. We can be guaranteed that if v is within range r from any reachable nodes in $A \cup V_M$, $R[v]$ will be set to $True$.

If there exists a path p that reaches t from s, it is possible to divide path p into x sub-paths such that (i) $(Head(p_i) \cup Tail(p_i)) \subset (A \cup S_{t+1})$ for $i \in [1, x]$, (ii) $Tail(p_i) = Head(p_{i+1})$ for $i \in [1, x-1]$, (iii) for any sub-path, its internal nodes are all on the same side of the separator. If we set $r = 2^s \geq \sum |p_x|$, then by the attribute of the *universal sequence* σ_s, we know that there is a sub-sequence from σ_s as $\langle c_1, c_2, ..., c_x\rangle$ that $|p_i| \leq c_i$ for $i \in [1, x]$. Thereby, $R[v]$ will be set to $True$ because of the induction. □

Proof of Space Complexity. In the algorithm, A is considered as a global variable and will take the most amount of space since its size will keep increasing with the depth of the recursion. If we calculated the upper bound of $|A|$, we have:

$$|A| \leq \sum_{i \in [1, t_{max}]} |S_i| + 2 \leq \sum_{i \in [1, t_{max}]} c_{sep}\sqrt{n_i} + 2$$

where $n_i \leftarrow |V_M|$ at depth i recursion. We know that when $|V_M| \geq 144c_{sep}^2$:

$$|V^b| \leq \frac{2}{3}|V| + c_{sep}\sqrt{|V|} \leq \frac{3}{4}|V|, b \in \{0,1\}$$

Thereby,

$$|A| \leq \sum_{i \in [1,t_{max}]} \sqrt{\left(\frac{3}{4}\right)^{i-1} n} + 2 \leq (c_{sep}\sqrt{n}) \sum_{i \geq 0} \left(\frac{3}{4}\right)^{i/2} + 2 = O(\sqrt{n})$$

□

Proof of Time Complexity. We let $N(t, 2^s)$ denote the number of recursion calls at depth t with $r = 2^s$. If we let t_{max} to be the maximum of depth for recursion, we have:

$$N(t, 2^s) = \begin{cases} 0 & t = t_{max} \\ 2 + 2N(t+1, 2^s) & s = 0, t < t_{max} \\ 2\sum_{i \in [1,2^{s+1}]}(1 + N(t+1, c_i)) & Otherwise \end{cases}$$

Hence, we get the following equations:

$$N(t, 2^0) \leq 2^{t_{max}-t+1}$$
$$N(t, 2^s) = 2N(t+1, 2^s) + 2N(t, 2^{s-1})$$

By induction, we can show that:

$$N(t, 2^s) \leq 2^{t_{max}-t+s+1}\binom{t_{max} - t + s}{s}$$

Therefore, the number of calls are polynomial bounded. Because we know that all processes for one recursion are within polynomial time, the total time upper bound is polynomial-time bounded.

□

7 Algorithm on Surface-Embedded Graphs

In this section, given an input graph G embedded on a surface S where every face is homeomorphic to an open disk, we introduce a reduction method that strongly follows [SV10]. If we let $A(x)$ represent a deterministic log-space algorithm that will output y such that $y \rightarrow x$ is an edge, we can define the forest decomposition of G with $F_A = \{(y, x) : x \in V(G) - \{s,t\} \cup \{source\ vertices\}, y = A(x)\}$. Thereby, each connected component will be a tree rooted at a source vertex, and the tree will be called a source tree. If we have vertices x and y in some source tree T, the tree curved at xy is the curve on S formed by the unique undirected path in T from x to y. For a given F_A and a path $P = \langle x_1, ..., x_k \rangle$ in G, P will be called irreducible if whenever there are $x_i, x_j \in F_A \cap P$, and $i < j$, P follows the tree edges in F_A from x_i, x_j.

If we have a closed curve C on S and removal of C disconnects S where at least one of the components is homeomorphic to a disk, curve C is defined as contractible. We denote an edge $x \rightarrow y$ to be local if (i) x and y are on the same source tree in F, (ii) the closed curve formed by xy and tree curve at xy are contractible, and (iii) there is no source on the interior surface. Otherwise, xy will be a global edge.

Definition 8. *We define two global edges $x \to y$ and $y \to z$ are topologically equivalent if the following two conditions are satisfied:*

- *They span the same source tree in F*
- *The closed curve in underlying undirected graph formed by edge xy, tree curve from y to z, edge zw, and tree curve from w to x bounds a connected portion of S, which we denoted as $D(xy, wz)$.*

Let E be an equivalence class of global edges that contains an edge e that spans two different source trees. We denote E_i to be the ith equivalence class that contains global edges that are lexicographically smaller than e and are lexicographically-least in their equivalence classes. Accordingly, we define the region enclosed by E_i as $R[E_i] = \cup_{e_1,e_2 \in E_i} D(e_1, e_2)$.

Definition 9. *We define the patterns to be the way that an irreducible path P shares the same edge with E_i. Specifically, every pattern consists of $\langle X, Y, Z\rangle$, where $X \in \{R, L\}$ represents whether the path enters E_i in left(L) or right(R) directions, $Z \in \{R, L\}$ stands for the directions that the path exits E_i, and $Y \in \{S, X\}$ represents whether the path enters and exits at the same(S) or different(X) ends. We denote N to be the full set of patterns.*

Note: Among all patterns, a path cannot enter and exist at the same end with different directions, which indicates that $\langle L, S, R\rangle$ and $\langle R, S, L\rangle$ are impossible. For the remaining six patterns, if the path enters and exists from different ends at the same direction, we call the pattern *nesting*. Otherwise, the pattern will be $full$.

Definition 10. *Given a DAG G, a pattern description is a tuple $x = (i, t, o, p)$, where $i \in \{1, ..., k\}, t \in \{A, B\}, o \in \{+1, -1\}, p \in N$. Specifically, i represents the index of equivalence class E_i; t represent the end of $R[E_i]$ that contains the entrance; o represents the orientation of the path with respect to the local orientation of the tree on the t-side of E_i; and p represents the pattern used in E_i. We denote V_p to be the set that contains all pattern descriptions.*

Note: For a given description x, we denote the incoming and outgoing edges in E_i with e_x^{in} and e_x^{out}.

Let $\mathcal{G}(m, g)$ denote the class of planar DAGs with at most m sources vertices embedded on a surface of genus at most g. Given a DAG G and its forest decomposition F, the pattern graph $P(V', E')$ is defined as follows. The vertex set $V' = \{s', t'\} \cup V_p = \{s', t'\} \cup (\{1, ..., k\} \times \{A, B\} \times \{+1, -1\} \times N)$. For two pattern descriptions $x, y \in V_p$, an edge $x \to y$ is in E' if and only if there exists an adjacency certificate with a list of nesting pattern descriptions $z_1, ..., z_l$, so that the following two conditions hold:

- There is an irreducible path from Head(e_x^{out}) to Tail(e_y^{in}) which induces the sequence $z_1, ..., z_l$ of nesting pattern descriptions.
- For each $j \in \{1, ..., l\}$, Tail($e_{z_j}^{in}$) is not reachable from Head(e_x^{out}) using irreducible paths that induce the pattern descriptions $z_1, ..., z_{j-1}$.

Moreover, for a description $x = (i, t_x, o, p)$ there is an edge $s' \to x$ in E'_p if and only if x has the t_x-end in the source tree of s. Also, there is an edge $x \to t'$ in E'_p if and only if the class E_i is incident to s, t_x is the other end of the class, and $p \in \{\langle RXL\rangle, \langle LXR\rangle\}$.

Theorem 11 ([SV10]). *There is a log-space reduction that, given an instance $< G, s, t >$ where $G \in \mathcal{G}(m, g)$ and s, t are vertices of G, outputs an instance $< G', s', t' >$ where G' is a directed graph and s', t' vertices of G', so that*

- *there is a directed path from s to t in G if and only if there is a directed path from s' to t' in G',*
- *G' has $O(m + g)$ vertices.*

Proof of Correctness. (Forward)If there exists an irreducible path P from s to t, P will induce a sequence of pattern descriptions $x_1, ..., x_l$ where there are edges from s to x_1 in the source tree of s. Thereby, there will be edges from s' to x_1. The same methods will be applied to $x_l \to t'$.

For a given pattern description of full type x centered at an edge class E_i and two vertices $y, z \notin R[E_i]$, we can prove that there is a path from y to z using only local paths if and only if there are local paths at the same direction as x from y to the entrance of x and from the exit of x to z. If we have full pattern x_{i+1} induced after x_i, we can show that $e^{in}_{x_{i+1}}$ is reachable by a local path from $e^{out}_{x_i}$. If the next full pattern after x_i is x_{i+j} where $j > 1$, we let $z_1, ..., z_{j-1}$ denote the intermediate nested patterns between x_i and x_{i+j}. If the nested patterns compose an adjacency certificate, then $x_i \to x_{i+j}$ exists in G'. Otherwise, there must be a $k \in [1, j-1]$ such that z_k violates the adjacency certificate. Assuming k is the smallest index among all patterns that break the adjacency, we know that there is an edge from x_i to z_k in G'. It can also be proved that $Tail(e^{in}_{x_j})$ is reachable from $Head(e^{out}_{z_k})$ by an irreducible path. Therefore, by several iterations, t' is reachable from s' in G' if there is a path from s to t in G.

(Backward) For a path $P = \langle s', x_1, ..., x_j, t' \rangle$ in G', with the similar methods, we can show that there is either a local path or an irreducible path from x_i to x_{i+1}, for $i \in [1, j-1]$. Since x_1 will be in the source tree of s', it is reachable by s. The same applies to the edge $x_j \to t'$ as well.

For the graph G with m sources embedded on a surface S with g genus, by Euler's formula, it can be proved that there are $O(m + g)$ equivalence classes of global edges. Since we add pattern descriptions by a constant multiple of equivalence classes, the number of vertices will be $O(m + g)$. □

Proof of Complexity. Given a pattern description x, we want to show that there is an algorithm that enumerates the pattern descriptions reachable by an edge in G' in log-space.

For a vertex v in a source tree, there is an ordering $E_{i_0}, E_{i_1}, ..., E_{i_l}$ of edge classes that are reachable by an irreducible path from v. For $j \in [1, l-1], E_{i_j}$ can be fully reached. Since v is in the interior of $R[E_{i_0}], E_{i_0}$ will not be fully reached. As a result, for each pattern y centered at E_{i_j} with $j \in [1, l-1]$, e^{in}_y is reachable from v. Hence, to enumerate all reachable patterns by x, we call the above methods with $v = e^{out}_x$ and add edges from x to all reachable patterns centered at E_{i_j}. E_{i_l} can be fully reached if we record the furthest edge and repeat the call another time. Therefore, the reduction will take a log-space. □

8 Open Problems

Apparently, solving STCON in polynomial time and $O(\log n)$ space remains the biggest challenge for the connectivity problems. In this survey, we discussed several algorithms for certain kinds of directed graphs. For the entire set of directed graphs, BBRS's algorithm still seems to be the most optimal solution.

Moreover, we think the attributes of unique-path graphs and reach-unambiguous graphs share significant similarities. If a broader definition can cover both kinds of graphs and a mix of both algorithms can handle all different cases, there might be an improvement of space upper bounds with a larger range of directed graphs.

9 Acknowledgement

This is the Senior Honors Thesis written by undergraduate Huanran Li at the University of Wisconsin-Madison, under the guidance of Professor Dieter van Melkebeek.

References

[AKNW14] Tetsuo Asano, David Kirkpatrick, Kotaro Nakagawa, and Osamu Watanabe. $\widetilde{O}(\sqrt{n})$ -space and polynomial-time algorithm for planar directed graph reachability. In *International Symposium on Mathematical Foundations of Computer Science*, pages 45–56. Springer, 2014.

[AL98] Eric Allender and K-J Lange. RUSPACE $(\log n) \subseteq$ DSPACE$(\log^2 n/\log\log n)$. *Theory of Computing Systems*, 31(5):539–550, 1998.

[AM04] Eric Allender and Meena Mahajan. The complexity of planarity testing. *Information and Computation*, 189(1):117, 2004.

[BBRS98] Greg Barnes, Jonathan F Buss, Walter L Ruzzo, and Baruch Schieber. A sublinear space, polynomial time algorithm for directed st connectivity. *SIAM Journal on Computing*, 27(5):1273–1282, 1998.

[INP$^+$13] Tatsuya Imai, Kotaro Nakagawa, Aduri Pavan, NV Vinodchandran, and Osamu Watanabe. An $o(n^{1/2+\varepsilon})$-space and polynomial-time algorithm for directed planar reachability. In *2013 IEEE Conference on Computational Complexity*, pages 277–286. IEEE, 2013.

[KKR08] Sampath Kannan, Sanjeev Khanna, and Sudeepa Roy. STCON in directed unique-path graphs. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[LT79] Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[Mil84] Gary L Miller. Finding small simple cycle separators for 2-connected planar graphs. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 376–382, 1984.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):1–24, 2008.

[SV10] Derrick Stolee and NV Vinodchandran. Space-efficient algorithms for reachability in surface-embedded graphs. Electonic Colloquium on Computational Complexity, 2010.