Depth-First Search in Directed Graphs, Revisited

Eric Allender
Rutgers University, USA
http://www.cs.rutgers.edu/~allender
allender@cs.rutgers.edu

Archit Chauhan
Chennai Mathematical Institute, India
https://www.cmi.ac.in/people/fac-profile.php?id=archit
archit.chauhan@gmail.com

Samir Datta
Chennai Mathematical Institute, India
https://www.cmi.ac.in/~sdatta/
sdatta@cmi.ac.in

Abstract

We present an algorithm for constructing a depth-first search tree in planar digraphs; the algorithm can be implemented in the complexity class UL, which is contained in nondeterministic logspace NL, which in turn lies in NC^2. Prior to this (for more than a quarter-century), the fastest uniform deterministic parallel algorithm for this problem was $O(\log^{10} n)$ (corresponding to the complexity class $\mathsf{AC}^{10} \subseteq \mathsf{NC}^{11}$).

We also consider the problem of computing depth-first search trees in other classes of graphs, and obtain additional new upper bounds.

2012 ACM Subject Classification Complexity Classes, Parallel Algorithms

Keywords and phrases Depth-First Search, Planar Digraphs, Parallel Algorithms, Space-Bounded Complexity Classes

Funding *Eric Allender*: Supported in part by NSF Grant CCF-1909216.
Archit Chauhan: Partially supported by a grant from Infosys foundation and TCS PhD fellowship.
Samir Datta: Partially supported by a grant from Infosys foundation and SERB-MATRICS grant MTR/2017/000480.

1 Introduction

Depth-first search trees (DFS trees) constitute one of the most useful items in the algorithm designer's toolkit, and for this reason they are a standard part of the undergraduate algorithmic curriculum around the world. When attention shifted to parallel algorithms in the 1980's, the question arose of whether NC algorithms for DFS trees exist. An early negative result was that the problem of constructing the *lexicographically least* DFS tree in a given digraph is complete for P [19]. But soon thereafter significant advances were made in developing parallel algorithms for DFS trees, culminating in the RNC^7 algorithm of Aggarwal, Anderson, and Kao [1]. This remains the fastest parallel algorithm for the problem of constructing DFS trees in general graphs, in the probabilistic setting, or in the setting of nonuniform circuit complexity. It remains unknown if this problem lies in (deterministic) NC (and we do not solve that problem here).

More is known for various restricted classes of graphs. For directed acyclic graphs (DAGs), the lexicographically-least DFS tree from a given vertex can be computed in AC^1 [9]. (See also [10, 7, 12, 18, 15, 14].) For undirected planar graphs, an AC^1 algorithm for DFS trees was presented by Hagerup [13]. For more general planar directed graphs Kao and Klein presented an AC^{10} algorithm. Kao subsequently presented an AC^5 algorithm for DFS in

strongly connected planar digraphs. In stating the complexity results for this prior work in terms of complexity classes (such as $\mathsf{AC}^1, \mathsf{AC}^{10}$, etc.), we are ignoring an aspect of this earlier work that was of particular interest to the authors of this earlier work: minimizing the number of processors. This is because our focus is on classifying the complexity of constructing DFS trees in terms of complexity classes. Thus, if we reduce the complexity of a problem from AC^{10} to NC^2, then we view this as a significant advance, even if the NC^2 algorithm uses many more processors (so long as the number of processors remains bounded by a polynomial). Indeed, our algorithms rely on the logspace algorithm for undirected reachability [20], which does not directly translate into a processor-efficient algorithm. We suspect that our approach can be modified to yield a more processor-efficient AC^1 algorithm, but we leave that for others to investigate.

1.1 Our Contributions

First, we observe that, given a DAG G, computation of a DFS tree in G logspace reduces to the problem of reachability in G. Thus, for general DAGs, computation of a DFS tree lies in NL, and for planar DAGs, the problem lies in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ [8, 22]. For classes of graphs where the reachability problem lies in L, so does the computation of DFS trees. One such class of graphs is planar DAGs with a single source (see [2], where this class of graphs is called SMPDs, for **S**ingle-source, **M**ultiple-sink, **P**lanar **D**AGs).

For undirected planar graphs, it was shown in [3] that the approach of Hagerup's AC^1 DFS algorithm [13] can be adapted in order to show that construction of a DFS tree in a planar *undirected* graph logspace-reduces to computing the distance between two nodes in a planar digraph. Since this latter problem lies in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ [23], so does the problem of DFS for planar *undirected* graphs.

Our main contribution in the current paper is to show that a more sophisticated application of the ideas in [13] leads to a $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ algorithm for construction of DFS trees in planar *directed* graphs. Since $\mathsf{UL} \subseteq \mathsf{NL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2$, this is a significant improvement over the best previous parallel algorithm for this problem: the AC^{10} algorithm of [17], which has stood for 27 years.

2 Preliminaries

We assume that the reader is familiar with depth-first search trees (DFS trees).

We further assume that the reader is familiar with the standard complexity classes L, NL and P (see e.g. the text [6]). We will also make frequent reference to the logspace-uniform circuit complexity classes NC^k and AC^k. NC^k is the class of problems for which there is a logspace-uniform family of circuits $\{C_n\}$ consisting AND, OR, and NOT gates, where the AND and OR gates have fan-in two and each circuit C_n has depth $O(\log^k n)$. (The logspace-uniformity condition implies that each C_n has only $n^{O(1)}$ gates.) AC^k is defined similarly, although the AND and OR gates are allowed unbounded fan-in. An equivalent characterization of AC^k is in terms of concurrent-read concurrent-write PRAMs with running time $O(\log^k n)$. For more background on these circuit complexity classes, see, e.g., the text [24].

A nondeterministic Turing machine is said to be *unambiguous* if, on every input x, there is at most one accepting computation path. If we consider logspace-bounded nondeterministic Turing machines, then unambiguous machines yield the class UL. A set A is in $\mathsf{co\text{-}UL}$ if and only if its complement lies in UL.

The construction of DFS trees is most naturally viewed as a *function* that takes a graph G and a vertex v as input, and produces as output an encoding of a DFS tree in G rooted at v. But the complexity classes mentioned above are all defined as sets of *languages*, instead of as sets of *functions*. Since our goal is to place DFS tree construction into the appropriate complexity classes, it is necessary to discuss how the complexity of functions fits into the framework of complexity classes.

When $\mathcal{C}$ is one of $\{\mathsf{L}, \mathsf{P}\}$, it is fairly obvious what is meant by "f is computable in $\mathcal{C}$"; the classes of logspace-computable functions and polynomial-time-computable functions should be familiar to the reader. However, the reader might be less clear as to what is meant by "f is computable in NL". As it turns out, essentially all of the reasonable possibilities are equivalent. Let us denote by FNL the class of functions that are computable in NL; it is shown in [16] each of the three following conditions is equivalent to "$f \in \mathsf{FNL}$".

1. f is computed by a logspace machine with an oracle from NL.
2. f is computed by a logspace-uniform NC^1 circuit family with oracle gates for a language in NL.
3. $f(x)$ has length bounded by a polynomial in $|x|$, and the set $\{(x, i, b) :$ the i^{th} bit of $f(x)$ is $b\}$ is in NL.

Rather than use the unfamiliar notation "FNL", we will abuse notation slightly and refer to certain functions as being "computable in NL".

The proof of the equivalence above relies on the fact that NL is closed under complement. Thus it is far less clear what it should mean to say that a function is "computable in UL" since it remains an open question if UL is closed under complement (although it is widely conjectured that $\mathsf{UL} = \mathsf{NL}$) [21, 5]). However the proof from [16] carries over immediately to the class $\mathsf{UL} \cap \mathsf{co\text{-}UL}$. That is, the following conditions are equivalent:

1. f is computed by a logspace machine with an oracle from $\mathsf{UL} \cap \mathsf{co\text{-}UL}$.
2. f is computed by a logspace-uniform NC^1 circuit family with oracle gates for a language in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$.
3. $f(x)$ has length bounded by a polynomial in $|x|$, and the set $\{(x, i, b) :$ the i^{th} bit of $f(x)$ is $b\}$ is in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$.

Thus, if any of those conditions hold, we will say that "f is computable in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$".

The important fact that the composition of two logspace-computable functions is also logspace-computable (see, e.g., [6]) carries over with an identical proof to the functions computable in L^C for any oracle C. Thus the class of functions computable in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ is also closed under composition. We make implicit use of this fact frequently when presenting our algorithms. For example, we may say that a colored labeling of a graph G is computable in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$, and that, given such a colored labeling, a decomposition of the graph into layers is also computable in logspace, and furthermore, that – given such a decomposition of G into layers – an additional coloring of the smaller graphs is computable in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$, etc. The reader need not worry that a logspace-bounded machine does not have adequate space to store these intermediate representations; the fact that the final result is also computable in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ follows from closure under composition. In effect, the bits of these intermediate representations are re-computed each time we need to refer to them.

3 DFS in DAGs logspace reduces to Reachability

In this section, we observe that constructing the lexicographically-least DFS tree in a DAG G can be done in logspace given an oracle for reachability in G. But first, let us define what we mean by the lexicographically-first DFS tree in G:

▶ **Definition 1.** *Let G be a DAG, with the neighbours of the vertices given in some order in the input. (For example, with adjacency lists, we can consider the ordering in which the neighbors are presented in the list). Then the lexicographic first DFS traversal of G is the traversal done by the following procedure:*

Input: (G, v)
Output: Sequence of edges in DFS tree
visited$[v] \leftarrow 1$
for *every out neighbour w of v, in the given order* **do**
  if *visited[w]* $= 0$ **then**
    print(v, w)
    $DFS(G, w)$
  end
end

Algorithm 1 Static DFS routine

That is, the lexicographically-first DFS tree is merely a DFS tree, but with the (very natural) condition that the children of every vertex are explored in the order given in the input.

When we apply this procedure as part of our algorithm for DFS in planar graphs, we will need to to apply it to directed acyclic *multigraphs* (i.e., graphs with parallel edges between vertices) where there is a logspace-computable function $f(v, e)$ that computes the ordering of the neighbors of vertex v, assuming that v is entered using edge e. (That is, if the DFS tree visits vertex v from vertex x, and there are several parallel edges from x to v, then the ordering of the neighbors of v may be different, depending on which edge is followed from x to v.)

As is observed in [9], the unique path from s to another vertex v in the lexicographically-least DFS tree in G rooted at s is the lexicographically-least path in G from s to t.

Now consider the following simple algorithm for constructing the lexicographically-least path in a DAG G from s to v, where:

Input: (G, s, v, f)
Output: Lex least path from s to v under f
$current \leftarrow s$; $e \leftarrow null$;
while $(current \neq v)$ **do**
  $child \leftarrow$ first child of $current$ (in the order given by $f(current, e)$)
  while $(REACH(child, v) \neq TRUE)$ **do**
    $child \leftarrow$ next child of $current$ (in the order given by $f(current, e)$)
  end
  $e \leftarrow (current, child)$; $current = child$;
end

Algorithm 2 DAG DFS routine

The correctness of this algorithm is essentially shown by the proof of Theorem 11 of [9].

The algorithm for computing the lexicographically-least DFS tree rooted at s can thus be presented as the composition of two functions g and h, where $g(G, s) = (G, s, L)$, where L is a list of all of the lexicographically-least paths from s to each vertex v. Note that the set of edges in the DFS tree in G rooted at s is exactly the set of edges that occur in the list L

in $g(G, s) = (G, s, L)$. Then $h(G, s, L)$ is just the result of removing from G each edge that does not appear in L. The function h is computable in logspace, whereas g is computable in logspace with an oracle for reachability in G.

Since reachability in DAGs is a canonical complete problem for NL, we obtain the following corollary:

▶ **Corollary 2.** *Construction of lexicographically-first DFS trees for DAGs lies in* NL.

Similarly, since reachability in planar directed (not-necessarily acyclic) graphs lies in UL ∩ co-UL [8, 22], we obtain:

▶ **Corollary 3.** *Construction of lexicographically-first DFS trees for planar DAGs lies in* UL ∩ co-UL.

A DAG G is said to be a SMPD if it contains at most one vertex of indegree zero. Reachability in SMPDs is known to lie in L [2].

▶ **Corollary 4.** *Construction of lexicographically-first DFS trees for SMPDs lies in* L.

3.1 DFS in a planar digraph with a single cycle

We now consider a special case that will form a useful subroutine for us: graphs in which there is a single cycle, forming the external face of the embedding. That is, let G be a planar digraph such that the external face is a directed cycle C and $G - V(C)$ is a DAG (or, alternatively, a directed acyclic *multi*graph). Then we can do DFS in G starting from an arbitrary vertex in C in UL ∩ co-UL. The DFS completes the cycle first and then, while backtracking, performs DFS in the reachable but as yet untraversed part of the digraph.

We now provide more details: Let the the vertices in the directed cycle C be $v_0, \ldots, v_k$, in this order, where the entry point in the cycle is v_0. Let $R(v_i) \subseteq V(G) \setminus V(C)$ be the set of vertices reachable from v_i in the graph excluding the cycle. We let $R'(v_i) = R(v_i) \setminus \bigcup_{j=i+1}^{k} R(v_j)$.

A logspace routine with an oracle for the UL ∩ co-UL problem of reachability in planar graphs can construct each of the sets $R'(v_i)$. It is clear that each $R'(v_i)$ induces a DAG which (if non-empty) consists of vertices reachable from v_i but not from subsequent v_j's. Moreover, the $R'(v_i)$'s are all pairwise disjoint. Thus a DFS of G can be performed by doing DFS on the graph induced by each $R'(v_i)$ (using Corollary 3) and unioning with the aforesaid DFS of the cycle C.

Note that a graph with a single cycle is a special case of a planar graph in which all cycles are clockwise (or all cycles are counterclockwise). By analogy with the *Coriolis effect*, we call such graphs *Coriolis graphs*. It turns out that Coriolis graphs play an important role in our main algorithm.

4 Layering the graph

The main algorithmic insight that led us to the current algorithm was an approach for finding DFS trees in Coriolis graphs. In the exposition below, we first layer the graph in terms of clockwise cycles (which we will henceforth call red cycles), and obtain a decomposition of the original graph into (essentially) Coriolis graphs. We then apply a nested layering in terms of counterclockwise cycles (which we will henceforth call blue cycles); ultimately we decompose the graph into units that are structured as a DAG, which we can then process using the tools from the earlier sections of the paper. The more detailed presentation follows.

4.1 Degree Reduction and Expansion

▶ **Definition 5.** *(of $\mathbf{Exp}^{\circlearrowleft}(G)$ and $\mathbf{Exp}^{\circlearrowright}(G)$) Let G be a planar digraph. The "expanded" digraph $\mathbf{Exp}^{\circlearrowleft}(G)$ (respectively, $\mathbf{Exp}^{\circlearrowright}(G)$ is formed by replacing each vertex v of total degree $d(v) > 3$ by a clockwise (respectively, counterclockwise) cycle C_v on $d(v)$ vertices such that the endpoint of the the i-th edge incident on v is now incident on the the i-th vertex of the cycle.*

$\mathbf{Exp}^{\circlearrowleft}(G)$ and $\mathbf{Exp}^{\circlearrowright}(G)$ each have maximum degree bounded by 3; i.e., they are *subcubic*. Next we define the clockwise (and counterclockwise) dual for such a graph and also a notion of distance.

Recall that for an undirected plane graph H, the dual (multigraph) H^* is formed by placing, for every edge $e \in E(H)$, a dual edge e^* between the face(s) on either side of e (see Section 4.6 from [11] for more details). Faces f of H and the vertices f^* of H^* correspond to each other as do vertices v of H and faces v^* of H^*.

▶ **Definition 6.** *(of Duals $G^{\circlearrowleft}$ and $G^{\circlearrowright}$) Let G be a plane digraph, then the clockwise dual $G^{\circlearrowleft}$ (respectively, counterclockwise dual $G^{\circlearrowright}$) is a weighted bidirected version of the undirected dual of the underlying undirected graph of G in a way so that the orientation formed by rotating the corresponding directed edge of G in a clockwise (respectively, counterclockwise) way gets a weight of 1 and the other orientation gets weight 0. We inherit the definition of dual vertices and faces from the underlying undirected dual.*

▶ **Definition 7.** *For a plane subcubic digraph G, let f_0 be the external face. Define the type $\mathbf{type}^{\circlearrowleft}(f)$ (respectively, $\mathbf{type}^{\circlearrowright}(f)$) of a face to be the singleton set consisting of the distance at which f lies from f_0 in $G^{\circlearrowleft}$: $\{d^{\circlearrowleft}(f_0, f)\}$ (respectively, $\{d^{\circlearrowright}(f_0, f)\}$). Generalise this to edges e by defining $\mathbf{type}^{\circlearrowleft}(e)$ (respectively $\mathbf{type}^{\circlearrowright}(e)$) as the set consisting of the union of the $\mathbf{type}^{\circlearrowleft}$ (respectively, $\mathbf{type}^{\circlearrowright}$) of the two faces adjacent to e, and to vertices v by defining as the $\mathbf{type}^{\circlearrowleft}(v)$ (respectively $\mathbf{type}^{\circlearrowright}(v)$) union of the $\mathbf{type}^{\circlearrowleft}$ (respectively, $\mathbf{type}^{\circlearrowright}$) of the faces incident on the vertex v.*

The following is a direct consequence of subcubicity and the triangle inequality:

▶ **Lemma 8.** *In every subcubic graph G, the cardinality $|\mathbf{type}^{\circlearrowleft}(x)|, |\mathbf{type}^{\circlearrowright}(x)|$ where x is a face, edge or a vertex is at least one and at most 2 and in the latter case consists of consecutive non-negative integers.*

Further, if $v \in V(G)$ is such that $|\mathbf{type}^{\circlearrowleft}(v)| = 2$, then there exist unique $u, w \in V(G)$, such that $(u, v), (v, w) \in E(G)$ and $|\mathbf{type}^{\circlearrowleft}(u, v)| = |\mathbf{type}^{\circlearrowleft}(v, w)| = 2$.

We first need a simple lemma:

▶ **Lemma 9.** *Suppose (f_1, f_2) is a dual edge with weight 1 (and (f_2, f_1) is of weight 0) then, $d^{\circlearrowleft}(f_0, f_1) \leq d^{\circlearrowleft}(f_0, f_2) \leq d^{\circlearrowleft}(f_0, f_1) + 1$.*

Proof. From the triangle inequality $d^{\circlearrowleft}(f_0, f_1) \leq d^{\circlearrowleft}(f_0, f_2) + d^{\circlearrowleft}(f_2, f_1) = d^{\circlearrowleft}(f_0, f_2)$. Similarly, $d^{\circlearrowleft}(f_0, f_2) \leq d^{\circlearrowleft}(f_0, f_1) + d^{\circlearrowleft}(f_1, f_2) \leq d^{\circlearrowleft}(f_0, f_1) + 1$. ◀

Proof. (of Lemma 8) Since each vertex $v \in V(G)$ of a subcubic graph is incident on at most 3 faces the only case is which $|\mathbf{type}^{\circlearrowleft}(v)| > 2$ corresponds to three distinct faces f_1, f_2, f_3 being incident on a vertex. But here the undirected dual edges form a triangle such that in the directed dual the 1 edges are oriented either as a cycle or acyclically. In the former case by three applications of the first half of Lemma 9 we get that $d^{\circlearrowleft}(f_0, f_1) \leq d^{\circlearrowleft}(f_0, f_2) \leq d^{\circlearrowleft}(f_0, f_3) \leq d^{\circlearrowleft}(f_0, f_1)$, hence all 3 distances are the same. Therefore $|\mathbf{type}^{\circlearrowleft}(v)| = 1$.

In the latter case, suppose the edges of weight 1 are $(f_1, f_2), (f_2, f_3), (f_1, f_3)$, then by Lemma 9 we get: $d^{\circlearrowleft}(f_0, f_1) \leq d^{\circlearrowleft}(f_0, f_2), d^{\circlearrowleft}(f_0, f_3) \leq d^{\circlearrowleft}(f_0, f_1) + 1$. Thus, both $d^{\circlearrowleft}(f_0, f_2), d^{\circlearrowleft}(f_0, f_3)$ are sandwiched between two consecutive values $d^{\circlearrowleft}(f_0, f_1), d^{\circlearrowleft}(f_0, f_1)+1$. Hence $d^{\circlearrowleft}(f_0, f_1)$, $d^{\circlearrowleft}(f_0, f_2)$, $d^{\circlearrowleft}(f_0, f_3)$ must take at most two distinct values, and thus $|\mathbf{type}^{\circlearrowleft}(v)| \leq 2$. Moreover either $\mathbf{type}^{\circlearrowleft}(f_1) \neq \mathbf{type}^{\circlearrowleft}(f_2) = \mathbf{type}^{\circlearrowleft}(f_3)$ or $\mathbf{type}^{\circlearrowleft}(f_1) = \mathbf{type}^{\circlearrowleft}(f_2) \neq \mathbf{type}^{\circlearrowleft}(f_3)$. Let e_1, e_2, e_3 be such that, ${e_1}^{\circlearrowleft} = (f_2, f_3), {e_2}^{\circlearrowleft} = (f_1, f_3), {e_3}^{\circlearrowleft} = (f_1, f_2)$. Then the two cases correspond to $|\mathbf{type}^{\circlearrowleft}(e_1)| = |\mathbf{type}^{\circlearrowleft}(e_2)| = 2, |\mathbf{type}^{\circlearrowleft}(e_3)| = 1$ and to $|\mathbf{type}^{\circlearrowleft}(e_1)| = 1, |\mathbf{type}^{\circlearrowleft}(e_2)| = |\mathbf{type}^{\circlearrowleft}(e_3)| = 2$ respectively. Noticing that e_1, e_3 are both incoming or both outgoing edges of v completes the proof for the clockwise case. The proof for the counterclockwise case is formally identical. ◀

▶ **Definition 10.** *For a plane subcubic graph G as above, we refer to vertices and edges with a type of cardinality two in $G^{\circlearrowleft}$ (respectively, in $G^{\circlearrowleft}$) as red (respectively, blue) while the ones with a cardinality of one as white. The resulting colored graphs are called* $\mathbf{red}(G)$ *and* $\mathbf{blue}(G)$ *respectively.*

We will see later how to apply both the duals in G to get red and blue layerings of a given input graph.

Also note that a red (respectively blue) edge must have red (respectively blue) end point vertices, as they are adjacent to the same faces as the edge between is.

We enumerate some properties of $\mathbf{red}(G)$, $\mathbf{blue}(G)$ (G is subcubic):

▶ **Lemma 11.** 1. *Red vertices and edges in* $\mathbf{red}(G)$ *form disjoint clockwise cycles.*
2. *No clockwise cycle in* $\mathbf{red}(G)$ *consists of only white edges(and hence white vertices). Similar properties hold for* $\mathbf{blue}(G)$.

Proof. 1. Firstly, note that a red edge must have red end point vertices, as they are adjacent to the same faces that the edge between them is adjacent to. It is immediate from Lemma 8 that if v is a red vertex, it has exactly one red incoming edge and one red outgoing edge, proving this part.
2. Suppose C is a clockwise cycle. Consider the shortest path in $G^{\circlearrowleft}$ from the external face to a face enclosed by C. From the Jordan curve theorem (Theorem 4.1.1 [11]), it must cross the cycle C. The edge dual to the crossing must be red.

◀

The definitions above, which apply only to subcubic plane graphs, can now be extended to a general plane graph G, by considering the subcubic graphs $\mathbf{Exp}^{\circlearrowleft}(G)$ (and $\mathbf{Exp}^{\circlearrowleft}(G)$). But first, we must make a simple observaion about $\mathbf{red}(\mathbf{Exp}^{\circlearrowleft}(G))$ (and dually about $\mathbf{blue}(\mathbf{Exp}^{\circlearrowleft}(G))$).

▶ **Lemma 12.** *Let $v \in V(G)$ be a vertex of degree more than 3. Let C_v be the corresponding expanded cycle in $\mathbf{Exp}^{\circlearrowleft}(G)$. Suppose at least one edge of C_v is white in $\mathbf{red}(\mathbf{Exp}^{\circlearrowleft}(G))$ then there is a unique red cycle C that shares edges with C_v.*

Proof. First we note that C_v does not contain anything inside it since it is an expanded cycle. By lemma 11 we know that C_v has at least one red edge. Suppose it shares one or more edges with a red cycle R_1. Since both cycles are clockwise and C_v has nothing inside, the cycle R_1 must enclose C_v. Now suppose there is another red cycle R_2 that shares one or more edges with C_v. Then R_2 must also enclose C_v. But two cycles cannot enclose a cycle whilst sharing edges with it without touching each other, which contradicts the above lemma that all red cycles in a subcubic graph are vertex disjoint. ◀

The last two lemmas allow us to consistently contract the red cycles in $\mathbf{red}(\mathbf{Exp}^{\circlearrowleft}(G))$:

▶ **Definition 13.** *The colored graph* $\mathbf{Col}^{\circlearrowleft}(G)$ *(respectively,* $\mathbf{Col}^{\circlearrowleft}(G)$*) is obtained by labeling a degree more than 3 vertex* $v \in V(G)$ *as red iff the cycle* C_v *in* $\mathbf{red}(\mathbf{Exp}^{\circlearrowleft}(G))$ *has at least one red edge and at least one white edge. Else the color of* v *is white. All the low degree vertices and edges of* G *inherit their colors from* $\mathbf{red}(\mathbf{Exp}^{\circlearrowleft}(G))$*. The coloring of* $\mathbf{Col}^{\circlearrowleft}(G)$ *is similar.*

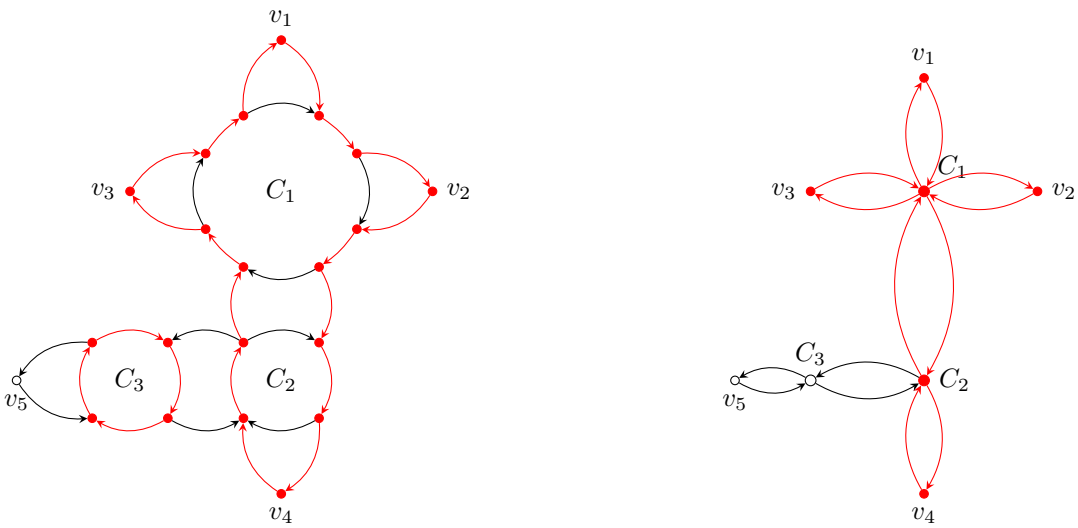


Figure 1 An example of contracting expanded cycles. The figure on right shows the graph after contracting the expanded cycles C_1, C_2, C_3 according to definition 13

We can now characterize the colorings in the graph $\mathbf{Col}^{\circlearrowleft}(G)$:

▶ **Lemma 14.** *The following hold:*

1. *A red cycle in* $\mathbf{Col}^{\circlearrowleft}(G)$ *is vertex disjoint from every red cycle contained in its interior.*
2. *Every 2-connected component of the red subgraph of* $\mathbf{Col}^{\circlearrowleft}(G)$ *is a simple clockwise cycle.*

Proof. For $v \in V(G)$, let $C_v \subseteq \mathbf{Exp}^{\circlearrowleft}(G)$ be the expanded cycle. If it has a red vertex it is immediately enclosed by a unique red cycle R in $\mathbf{Exp}^{\circlearrowleft}(G)$ by Lemma 12. Assuming C_v is not all red, it consists of alternating red subpaths and white subpaths. On contracting C_v we get a collection of clockwise red cycles outside sharing a common cut-vertex v. Notice that the new collection of red cycles consists of edges that R did not share with C_v. Also notice that (as a thought experiment) if we contracted the C_v's that share a vertex with R, one at a time we would get an edge-disjoint set of red cycles with distinct cut vertices. Therefore, in $\mathbf{Col}^{\circlearrowleft}(G)$, the red subgraph consists of a collection of connected components, each of which is a remnant of exactly one red cycle in $\mathbf{Exp}^{\circlearrowleft}(G)$; these connected components consist of red cycles that touch externally at cut vertices. Hence both parts of the lemma follow. ◀

Although the above lemmas have been proved for the clockwise dual, they also hold for counter clockwise dual with red replaced by blue mutatis mutandis.

4.2 Layering the colored graphs

▶ **Definition 15.** *Let* $x \in V(\mathbf{Col}^{\circlearrowleft}(G)) \cup E(\mathbf{Col}^{\circlearrowleft}(G))$*. Let* $\ell^{\circlearrowleft}(x)$ *be one more than the minimum integer that occurs in* $\mathbf{type}^{\circlearrowleft}(x')$*, for each* $x' \in V(\mathbf{Exp}^{\circlearrowleft}(G)) \cup E(\mathbf{Exp}^{\circlearrowleft}(G))$ *that*

is contracted to x. Further let $\mathcal{L}^k(\mathbf{Col}^{\circlearrowleft}(G)) = \{x \in V(\mathbf{Col}^{\circlearrowleft}(G)) \cup E(\mathbf{Col}^{\circlearrowleft}(G)) : \ell^{\circlearrowleft}(x) = k\}$. Similarly define, $\ell^{\circlearrowleft}(x), \mathcal{L}^k(\mathbf{Col}^{\circlearrowleft}(G))$.

We call $\mathcal{L}^k(\mathbf{Col}^{\circlearrowleft}(G))$ the k^{th} layer of the graph.

It is easy to see the following from Lemma 14:

▶ **Proposition 16.** *For every $x \in V(\mathbf{Col}^{\circlearrowleft}(G)) \cup E(\mathbf{Col}^{\circlearrowleft}(G))$ the quantity $\ell^{\circlearrowleft}(x)$ is one more than the number of red cycles that strictly enclose x in $\mathbf{Col}^{\circlearrowleft}(G)$. All the vertices and edges of a red cycle of $\mathbf{Col}^{\circlearrowleft}(G)$ lie in the same layer $\mathcal{L}^{k+1}(\mathbf{Col}^{\circlearrowleft}(G))$ for the enclosure depth k of the cycles.*

We had already noted above that the red subgraph of G had simple clockwise cycles as its biconnected components. We note a few more lemmas about the structure of a layer of G:

▶ **Lemma 17.** *We have:*

1. *A red cycle in a layer $\mathcal{L}^{k+1}(\mathbf{Col}^{\circlearrowleft}(G))$ does not contain any vertex/edge of the same layer inside it.*
2. *Any clockwise cycle in a layer consists of all red vertices and edges.*

Dually, a blue cycle in a layer does not contain any vertex or edge of the same layer inside it.

▶ Remark 18. Notice that the conclusion in the second part of the Lemma fails to hold if we allow cycles spanning more than one layer.

Proof. The first part is a direct consequence of proposition 16. For the second part we mimic the proof of the second part of Lemma 11. Consider a clockwise cycle $C \subseteq \mathcal{L}^{k+1}\mathbf{Col}^{\circlearrowleft}G$ that passes through a white edge e. Every face adjacent to C from the outside must have $\mathbf{type}^{\circlearrowleft} = k$ because C is contained in layer $k+1$. Then the $\mathbf{type}^{\circlearrowleft}$ of the faces on either side of e is the same and therefore must be k. Let f be a face enclosed by C that has $\mathbf{type}^{\circlearrowleft}(f) = k$. Thus it must be adjacent to a face of $\mathbf{type}^{\circlearrowleft} = k-1$. But this contradicts that every face inside and adjacent to C must have $\mathbf{type}^{\circlearrowleft}$ at least k. ◀

The above lemmas show that the strongly connected components of the red subgraph of a layer consist of red cycles touching each other without nesting, in a tree like structure. This prompts the following definition:

▶ **Definition 19.** *For a red cycle $R \subseteq \mathcal{L}^k(\mathbf{Col}^{\circlearrowleft}(G))$ we denote by G_R, the graph induced by vertices of $\mathcal{L}^{k+1}(\mathbf{Col}^{\circlearrowleft}(G))$ enclosed by R.*

The strongly connected components of the red subgraph of G_R are called the red clusters of G_R.

The cluster graph $\mathbf{Cl}^{\circlearrowleft}(G_R)$ is formed from G_R by contracting the red clusters of G_R to single nodes along with all the white vertices of G_R and adding a directed edge between two nodes iff there was a directed edge between corresponding vertices in G_R.

We get:

▶ **Lemma 20.** *For each red cycle $R \subseteq \mathcal{L}^k(\mathbf{Col}^{\circlearrowleft}(G))$, the cluster graph $\mathbf{Cl}^{\circlearrowleft}(G_R)$ does not contain any clockwise cycle. That is, it is a Coriolis graph.*

Proof. If there is a clockwise cycle $C \subseteq \mathbf{Cl}^{\circlearrowleft}(G_R)$ then there must be a corresponding clockwise cycle $C' \subseteq G_R$ as well. It cannot be all red since otherwise it would map to a single vertex in $\mathbf{Cl}^{\circlearrowleft}(G_R)$. But this contradicts Lemma 17. ◀

Next we aim to remove all the counterclockwise cycles in order to construct a DAG in which we can do DFS. For this we apply another layering on every layer $\mathcal{L}^k(\mathbf{Col}^{\circlearrowleft}(G))$ of the graph G again with the help of Definitions 13, 15, but this time using counterclockwise i.e. blue cycles. Thus for every red cycle R in G, we consider the graph $H = \mathbf{Col}^{\circlearrowleft}(G_R)$ and its layers $\mathcal{L}^l(H)$(w.r.t the counterclockwise dual) for non-negative integers l. Consider a blue cycle $B \subseteq \mathcal{L}^l(H)$ and consider the corresponding blue graph H_B. By Lemma 20 applied in a counterclockwise sense, there is no counterclockwise cycle in the cluster graph $\mathbf{Cl}^{\circlearrowleft}(H_B)$.

The lemmas above about the structure of a red layer also hold for a blue layer with suitable changes.

It turns out that if we compress the strongly connected components of the colored subgraph (both red *and* blue) of a blue layer, we get a DAG.

Formally, we start with the combined analog of Definitions 13, 15:

▶ **Definition 21.** *Each vertex or edge $x \in V(G) \cup E(G)$ gets a red layer number $k+1$ if it belongs to $\mathcal{L}^{k+1}(\mathbf{Col}^{\circlearrowright}(G))$ and a blue layer number $l+1$, if it belongs to $\mathcal{L}^{l+1}(\mathbf{Col}^{\circlearrowleft}(G_R))$ where $R \subseteq \mathcal{L}^k(\mathbf{Col}^{\circlearrowright}(G))$ is the red cycle immediately enclosing x.*

Moreover this defines the colored graph $\mathbf{Col}(G)$ by giving x the color red if it is red in $\mathbf{Col}^{\circlearrowright}(G)$ and/or blue in $\mathbf{Col}^{\circlearrowleft}(G_R)$ (notice it could be both red and blue) and lastly white if it is white in both the graphs. In this case, we say that x belongs to sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$.

By proposition 16, we can also say that a sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$ thus consists of edges/vertices that are strictly enclosed inside k red cycles and inside l blue cycles that are contained *inside* the first enclosing red cycle.

We'll see some observations and lemmas regarding the structure of a sublayer now.

Since every edge/vertex in $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$ has the same red AND blue layer number, it is clear that there can be no nesting of colored cycles. Also we have:

▶ **Lemma 22.** *Every clockwise cycle in a sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$ consists of all red edges and vertices and any every counterclockwise cycle in the sublayer consists of all blue vertices and edges. (Some edges/vertices of the cycle can be both red as well as blue)*

Proof. This is a direct consequence of Lemma 17 applied to the sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$, which is a (counterclockwise) layer in graph G_R for some red cycle R. ◀

Thus we can refer to clockwise cycles and counterclockwise cycles as red and blue cycles respectively.

▶ **Definition 23.** *For a red or blue colored cycle C of layer $\mathcal{L}^{k,l}(\mathbf{Col}(G))$, we denote by G_C the graph induced by vertices of $\mathcal{L}^{k',l'}(\mathbf{Col}(G))$ enclosed by C, where $\{k', l'\}$ is $\{k+1, 1\}$ or $\{k, l+1\}$ according to whether C is red or blue cycle respectively.*

Note that two cycles of the same color in $\mathcal{L}^{k+1,l+1}(G)$ cannot share an edge since neither is enclosed by the other – since they belong to the same layer, and they also have the same orientation. Cycles of different colors can share edges but we note:

▶ **Lemma 24.** *Two cycles of a sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$ can only share one contiguous segment of edges.*

Proof. Let a red cycle R and a blue cycle B in a sublayer sublayer share two vertices u, v but let the paths $R(u,v), B(u,v)$ in the two cycles be disjoint. Notice that the graph $(R \setminus R(u,v)) \cup B(u,v)$ is also a clockwise cycle that encloses the edges of $R(u,v)$ contradicting the first part of Lemma 17. ◀

We consider the strongly connected components of a sublayer and note the following lemmas regarding them:

▶ **Lemma 25.** *The strongly connected components of a sublayer, which we call* clusters*, have the following properties:*

1. *Every vertex/edge in them is either blue or red (or possibly both).*
2. *Every face is a directed cycle(red or blue).*

Proof. 1. In a strongly connected graph every vertex and edge lies on a cycle and therefore by Lemma 22 must be colored red or blue (or both).

2. Suppose there is a face f the boundary of which is not a directed cycle. Look at a directed dual (say clockwise) of the strongly connected component (just the component independently). This dual must be a DAG since the primal is strongly connected. The vertex f^* in the dual corresponding to face f of the strongly connected component has in degree at least one and out degree at least one since it has boundary edges of both orientations. Consider a vertex u^* of the dual which has an edge (u^*, f^*) to f^*. If we contract this edge, merging u^* and f^* to f^*, the modified dual is still a DAG clearly, with one less vertex. If we keep merging the vertices incident to f^* into it, eventually we must reach a stage when no vertex is incident to f^*. This merged f^* is a source since its in-degree is 0, and hence its outgoing edges form a directed cut for this modified dual. But this also clearly corresponds to a directed cut in the original dual, with one partition containing the dual vertex f^* and all other dual vertices that were merged with f^*. In the primal, by cut cycle duality this corresponds to a directed cycle that contains the face f and the faces corresponding to the dual vertices merged with f^*. Thus cycle thus contains more than one face inside it along with f, which violates lemma 22 since directed cycles are empty for that sublayer.

◀

The strongly connected components or clusters of a sublayer hence consist of intersecting red and blue cycles. However they can only intersect in a tree like manner as we will see from following definition and lemma.

We now construct the incidence graph of these strongly connected components. In other words,

▶ **Definition 26.** *The nodes of the graph $\mathbf{S}^{k+1,l+1}(G)$ are the directed cycles of each of the two colors (viz. red and blue) in the layer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$. Two nodes support an undirected edge if the corresponding strongly connected components intersect.*

We have the following:

▶ **Lemma 27.** *$\mathbf{S}^{k+1,l+1}(G)$ is a forest. Given an entry point into a component of $\mathbf{S}^{k+1,l+1}(G)$ we can, in* L*, compute the DFS of such a tree.*

Proof. For any two cycles C_1, C_2 that are adjacent and any $v_1 \in C_1, v_2 \in C_2$ it is the case that there is a directed path in the sublayer from v_1 to v_2 (via $V(C_1) \cap V(C_2)$); thus inductively the same property holds for any two C_1, C_2 in the same connected component of $\mathbf{S}^{k+1,l+1}(G)$. Since $\mathbf{S}^{k+1,l+1}(G)$ is a planar (undirected) graph, it follows that if it is not a forest, then it must enclose a facial cycle f. This facial cycle f corresponds to a face f' in the sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$. Each node on the boundary of f corresponds to a directed cycle in $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$, and the face f' must be incident on each of these cycles. By Lemma 25, f' must be a red cycle or a blue cycle. Without loss of generality, suppose it is red. But this means that it cannot intersect a red cycle corresponding to a node on the

boundary of f in more than a vertex. Thus it consists exclusively of edges from some blue cycles, call them $B_1, \ldots, B_k$, on the boundary of f. Thus f' is entirely enclosed by the edges of $\cup_{i=1}^{k} E(B_i) \setminus E(f')$, which form a blue cycle. This contradicts Lemma 17. ◀

Next,we extend the definition of cluster graphs (Definition 19) by contracting the clusters, which are maximal trees of the forest $\mathbf{S}^{k+1,l+1}(G)$ to single vertices:

▶ **Definition 28.** *Consider the multigraph $\mathbf{Cl}^{k+1,l+1}(G)$ on the vertex set $V(\mathbf{Cl}(G)) = \{v_T : T$ is a maximal tree in $\mathbf{S}^{k+1,l+1}(G)\} \cup \{v : v \in V(H)$ is colored white in $\mathbf{Col}(G)\}$; each edge in G is carried over to $\mathbf{Cl}^{k+1,l+1}(G)$, resulting in parallel edges when vertices in G are merged into a single vertex in $\mathbf{Cl}^{k+1,l+1}(G)$.*

Thus, we obtain the following:

▶ **Lemma 29.** *$\mathbf{Cl}^{k+1,l+1}(G)$ is a directed acyclic multigraph for every $k, l \geq 0$.*

Proof. Trivial since clusters are the strongly connected components of the sublayer. ◀

5 The Algorithm for DFS in a Planar Graph

Now we will use the layering and lemmas from the previous section to give the final algorithm for DFS in a general planar digraph H, from a root r. Our output will consist of the edges that are included in the DFS tree, along with an ordering on the outgoing tree edges for every vertex in the graph, since – in contrast to the case for undirected DFS trees – a directed spanning tree may or may not be a DFS tree for different traversals. The ordering on outgoing edges for every vertex fixes the traversal.

The first step is to build the graph $G \subseteq H$ consisting of all vertices that are reachable from r, which can be done in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$. A planar embedding of G with r on the external face f_0 can then be constructed, using logarithmic space [4, 20].

To help make the indexing of our layers simpler, create a "dummy" red cycle (essentially just a self loop on a "pseudo-root vertex" r_0' with an edge from r_0 to r, where the self loop completely encloses G; this has the effect of placing the root r in layer 1.

Note that the labeling of G (described in the previous section) can be computed in logspace with an oracle for computing distance in planar graphs. This is because the type of each face, edge, and vertex is given by computing distances in the dual graph. Computing distance in planar graphs lies in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$ [23, Section 4], and thus computing $\mathbf{Col}(G)$ can be done in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$.

With $\mathbf{Col}(G)$ in hand, we define a meta tree of the laminar family of colored cycles of G.

▶ **Definition 30.** *For a planar digraph G, with red and blue cycles given by $\mathbf{Col}(G)$, the meta tree T_G is an undirected tree with nodes representing the colored cycles of G. The root node of T_G is the self-loop on r_0 belonging to sublayer $\mathcal{L}^{0,1}(\mathbf{Col}(G))$. For a node in T_G representing cycle C of a sublayer $\mathcal{L}^{k+1,l+1}(\mathbf{Col}(G))$, its children are the cycles of the next sublayer that are contained inside C.*

Note that every node of G appears in some subgraph $\mathbf{S}^{k+1,l+1}$ inside some colored cycle C of $\mathbf{Col}(G)$). First, we describe how to process the subgraph $C \cup \mathbf{S}^{k+1,l+1}$, and then we describe the order in which we process the colored cycles (which will also determine the vertex v in which we first enter the cycle C).

Note that the multigraph consisting of C along with the directed acyclic multigraph $\mathbf{Cl}^{k+1,l+1}(G)$ contained in C is precisely the sort of graph that we showed how to search in

Section 3.1. A DFS of this graph can be performed in $\mathsf{UL} \cap \mathsf{co\text{-}UL}$. But many of the nodes of $\mathbf{S}^{k+1,l+1}$ are not simply nodes of G, but are clusters of cycles in G. Thus we must output a DFS not of $\mathbf{Cl}^{k+1,l+1}(G)$ but a DFS of the corresponding nodes in G.

1. Start the DFS of $C\cup$ the multigraph $\mathbf{Cl}^{k+1,l+1}(G)$ that lies within C, as described in Section 3.1, by following the edges of C until we come back to the entry vertex v.
2. Then start backtracking along C and performing a DFS of the directed acyclic multigraph $\mathbf{Cl}^{k+1,l+1}(G)$. Each time we follow an edge to a new vertex D of $\mathbf{Cl}^{k+1,l+1}(G)$ that represents a cluster of G, this edge corresponds to an edge e of G to a node x on one of the cycles of the undirected tree of cycles that constitutes the cluster D. The ordering of the neighbors of D (that is used in constructing the lexicographic-least DFS tree of $\mathbf{Cl}^{k+1,l+1}(G)$) consists of the order in which edges out of D are encountered while searching the tree of cycles that constitutes D, when starting at vertex x; this ordering can be computed in logspace.
3. Each vertex of D of $\mathbf{Cl}^{k+1,l+1}(G)$ represents a tree of cycles. Each cycle in the cluster is explored by going from its entry vertex directly around the cycle, and then backtracking to explore its neighbor cycles in the cluster. This is easy to perform in logspace. (This sequence of exploring the cycles in D imposes the order on the edges that leave D to other clusters in $\mathbf{Cl}^{k+1,l+1}(G)$, which gives us the ordering that determines the lexicographically-least DFS tree of $\mathbf{Cl}^{k+1,l+1}(G)$.)
4. The lexicographically-least DFS tree of $\mathbf{Cl}^{k+1,l+1}(G)$ identifies the edge that should be used to visit each neighbor of D. Explore each vertex of $\mathbf{Cl}^{k+1,l+1}(G)$ in turn in this way.

And now we describe the algorithm that determines the order in which we process the colored cycles. For each node C in the meta tree T_G, (and recall that each node in T_G corresponds to a colored cycle), find the unique path in T_G from the root to C. Then start following that path; for each edge $C_1 \to C_2$ in that path, we start by knowing the vertex v in C_1 where the tree constructed thus far entered C_1. (Initially, C is the self loop on r_0, and $v = r_0$.)

Follow the procedure outlined above for processing the DFS tree inside of C_1, but do not produce any output. Instead, wait for the moment when C_2 is encountered in that process. (It will be encountered, because otherwise there would not be an edge $C_1 \to C_2$ in the meta tree.) At that point, remember the vertex x where cycle C_2 is first entered, and then start processing the next edge in the path from the root to C.

When C is finally reached, we remember the vertex where C was entered, and start outputting the DFS tree for the subgraph inside C, as above.

We must also give the orderings of outgoing tree edges around every vertex. For a white vertex of any sublayer, the outgoing edges belong to the same sublayer and their ordering is already defined by the algorithm in section 3.1. For the other case, we analyze:

Suppose v is a vertex on a colored cycle C of some sublayer. Let the outgoing tree edges be e, $e'_1, e'_2 ... e'_k$, $e''_1, e''_2 ... e''_l$, where e is the outgoing edge that belongs to cycle C, $e'_1, e'_2 ... e'_k$ are the outgoing edges other than e that belong to the same layer as v (they consist of edges going out of C, either white edges going out from the cluster or colored edges of the same cluster), and $e''_1, e''_2 ... e''_l$ are the outgoing edges of the *next* layer (edges going inside of C).

Then the order of these edges for DFS is:

- First we take the white edges among $e'_1 ... e'_k$.
- Then we take e(finish the cycle).
- Then we take the colored outgoing edges among $e'_1 ... e'_k$.
- Then we take the edges of the next layer, $e''_1 ... e''_l$.

The order of edges *within* each of these steps is already defined in section 3.1 or in steps 2 and 3 of the algorithm above. This gives an ordering of all outgoing tree edges for any vertex v.

We could interchange between last two points of the ordering, and still the algorithm would give DFS trees albeit different ones. However it is crucial that e is taken before $e''_1 ... e''_l$ for all vertices, i.e. we finish the cycle before going inside to higher sublayers.

The algorithm clearly can be implemented in logspace with an oracle for $\mathsf{UL} \cap \mathsf{co\text{-}UL}$, and it clearly outputs a tree that spans G.

Now we must show that the tree that is produced is a DFS tree.

Our algorithm definitely produces a spanning tree of the set of all vertices in G that are reachable from the start vertex r. In order to show that the tree is a DFS tree, it suffices to show that, for any edge (u, v) of G that is not in the tree, in our depth-first traversal of the tree the vertex v is visited before u, or else v is visited from a descendent of u in the tree.

Either u and v are in the same level, or else u and v are at different levels.

- Case 1: u and v are in the same sublayer:
  Then either u and v are in the same cluster, or they are not. If they are not in the same cluster, then the cluster that v is in is visited by some lexicographically-earlier edge from the cluster in which u resides. Thus v is visited before u in the depth-first traversal of the tree.
  If u and v are in the same cluster, then either they in the same colored cycle, or they are not. If they are in the same colored cycle, and the edge (u, v) is not in the tree, it can either be because v is the first vertex visited in the cycle, and thus v is visited before u, or else edge (u, v) is a chord of the cycle containing u, v (but the chord itself is in the next sublayer by definition). Since we traverse the cycle first and then branch inside, edge (u, v) is either a forward edge or a back edge depending on whether u comes first in cycle or v.
  If u and v are in different colored cycles in the same cluster, then there is not an edge (u, v).
- Case 2: v is in a higher sublayer than u
  In this case u must be on a colored cycle C and v lies inside C, in the next sublayer. Since in our algorithm we complete the traversal of cycle C first and then explore the clusters inside, the only way (u, v) can be a non tree edge is when v has been explored in the subtree of a vertex u' that occurs after u in traversal of C, while backtracking. The edge (u, v) is therefore a forward edge.
- Case 3: u is in a higher sublayer than v
  This case is similar to previous one and the same argument shows that v must be on a colored cycle and the edge (u, v) is a back edge.

Thus our tree is a DFS tree.

References

1 Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990. doi:10.1137/0219025.

2 Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4):675–723, 2009. doi:10.1007/s00224-009-9172-z.

3 Eric Allender, Archit Chauhan, Samir Datta, and Anish Mukherjee. Planarity, exclusivity, and unambiguity. *Electronic Colloquium on Computational Complexity (ECCC)*, 26:39, 2019.

4 Eric Allender and Meena Mahajan. The complexity of planarity testing. *Inf. Comput.*, 189:117–134, 2004.

5 Eric Allender, Klaus Reinhardt, and Shiyu Zhou. Isolation, matching, and counting: Uniform and nonuniform upper bounds. *Journal of Computer and System Sciences*, 59(2):164–181, 1999.

6 Sanjeev Arora and Boaz Barak. *Computational Complexity, a modern approach*. Cambridge University Press, 2009.

7 Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-first search using $O(n)$ bits. In Hee-Kap Ahn and Chan-Su Shin, editors, *Proc. 25th International Symposium on Algorithms and Computation (ISAAC)*, volume 8889 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2014. doi:10.1007/978-3-319-13075-0_44.

8 Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. *TOCT*, 1(1):4:1–4:17, 2009. URL: http://doi.acm.org/10.1145/1490270.1490274, doi:10.1145/1490270.1490274.

9 Pilar de la Torre and Clyde P. Kruskal. Fast parallel algorithms for all-sources lexicographic search and path-algebra problems. *J. Algorithms*, 19(1):1–24, 1995. doi:10.1006/jagm.1995.1025.

10 Pilar de la Torre and Clyde P. Kruskal. Polynomially improved efficiency for fast parallel single-source lexicographic depth-first search, breadth-first search, and topological-first search. *Theory Comput. Syst.*, 34(4):275–298, 2001. doi:10.1007/s00224-001-1008-4.

11 Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate texts in mathematics*. Springer, 2016.

12 Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient basic graph algorithms. In *Proc. 32nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 30 of *LIPIcs*, pages 288–301. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.288.

13 Torben Hagerup. Planar depth-first search in O(log n) parallel time. *SIAM J. Comput.*, 19(4):678–704, June 1990. URL: http://dx.doi.org/10.1137/0219047, doi:10.1137/0219047.

14 Torben Hagerup. Space-efficient DFS and applications to connectivity problems: Simpler, leaner, faster. *Algorithmica*, 82(4):1033–1056, 2020. doi:10.1007/s00453-019-00629-x.

15 Taisuke Izumi and Yota Otachi. Sublinear-space lexicographic depth-first search for bounded treewidth graphs and planar graphs. In *Proc. 47th International Colloquium on Automata, Languages and Programming (ICALP)*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. to appear.

16 B. Jenner and B. Kirsig. Alternierung und Logarithmischer Platz. Dissertation, Universität Hamburg, 1989.

17 Ming-Yang Kao and Philip N. Klein. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. *Journal of Computer and System Sciences*, 47(3):459–500, 1993. doi:10.1016/0022-0000(93)90042-U.

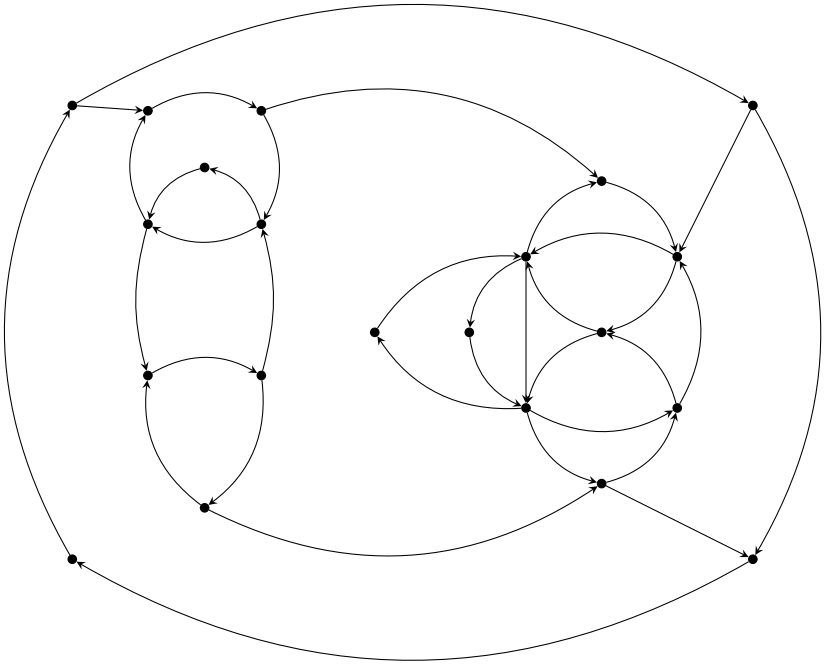
18 Maxim Naumov, Alysson Vrielink, and Michael Garland. Parallel depth-first search for directed acyclic graphs. In *Proc. 7th Workshop on Irregular Applications: Architectures and Algorithms*, pages 4:1–4:8, 2017. doi:10.1145/3149704.3149764.

19 John H. Reif. Depth-first search is inherently sequential. *Inf. Process. Lett.*, 20(5):229–234, 1985. doi:10.1016/0020-0190(85)90024-9.

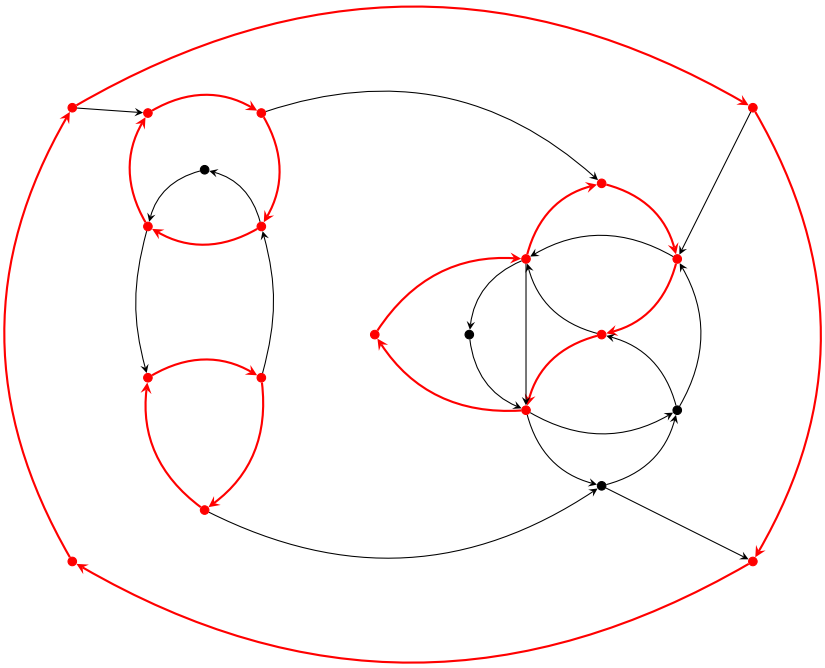
20 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), 2008.

21 Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, 2000. URL: https://doi.org/10.1137/S0097539798339041, doi:10.1137/S0097539798339041.

626 **22** Raghunath Tewari and N. V. Vinodchandran. Green's theorem and isolation in planar
627 graphs. *Inf. Comput.*, 215:1–7, 2012. URL: https://doi.org/10.1016/j.ic.2012.03.002,
628 doi:10.1016/j.ic.2012.03.002.
629 **23** Thomas Thierauf and Fabian Wagner. The isomorphism problem for planar 3-connected
630 graphs is in unambiguous logspace. *Theory Comput. Syst.*, 47(3):655–673, 2010. doi:10.1007/
631 s00224-009-9188-4.
632 **24** H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New
633 York Inc., 1999. doi:10.1007/978-3-662-03927-4.



(a)



(b)

Figure 2 Figure (a) is a graph G. Figure (b) is the graph in (a) after labelling red edges using clockwise dual. We omit the cycle expansion and contraction procedure here.

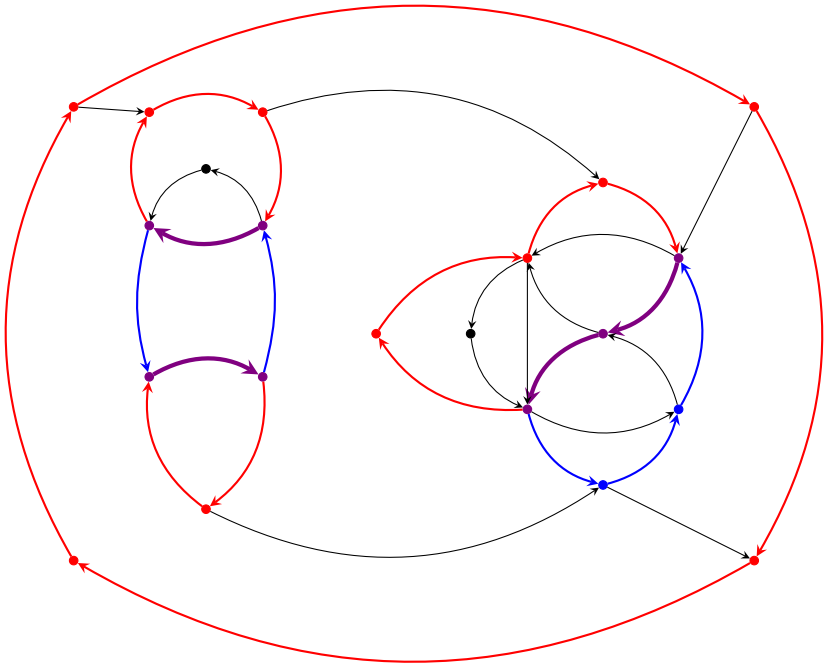


Figure 3 This figure shows G after applying blue labellings to each red layer we obtained in the previous figure. The vertices and edges colored purple are those that are red as well as blue.

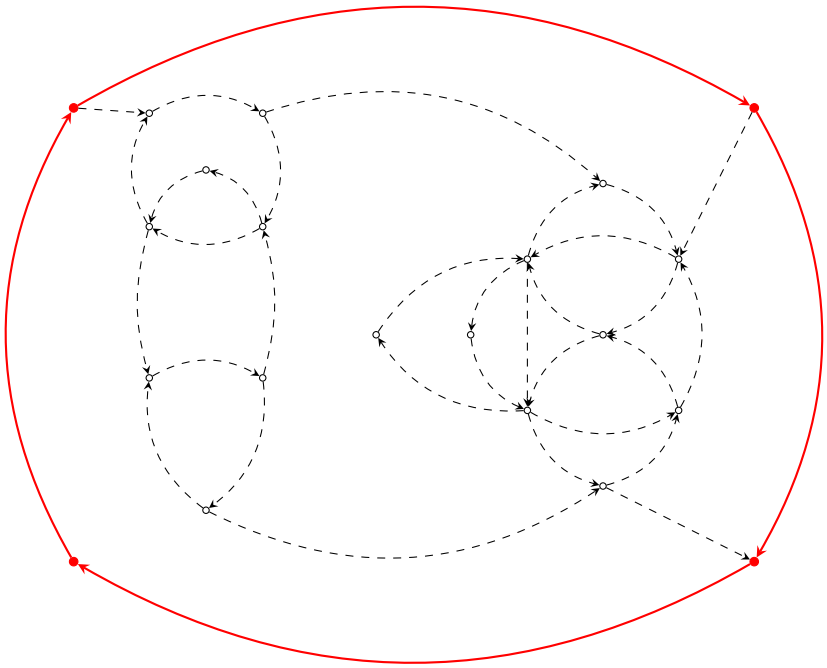


Figure 4 This figure represents the sublayer $(1, 1)$. The dashed edges and empty vertices are not part of the layer.

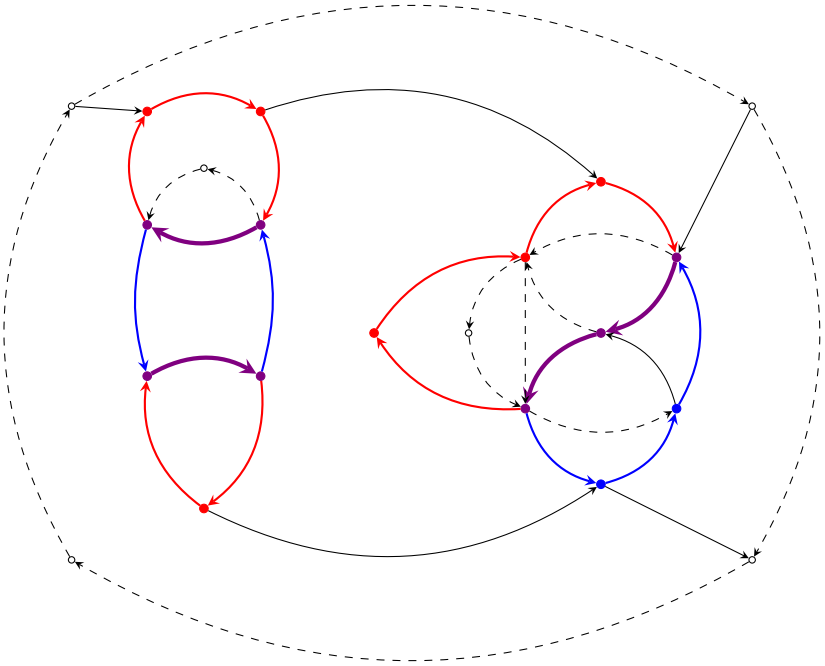


Figure 5 This figure represents the sublayer $(2, 1)$.

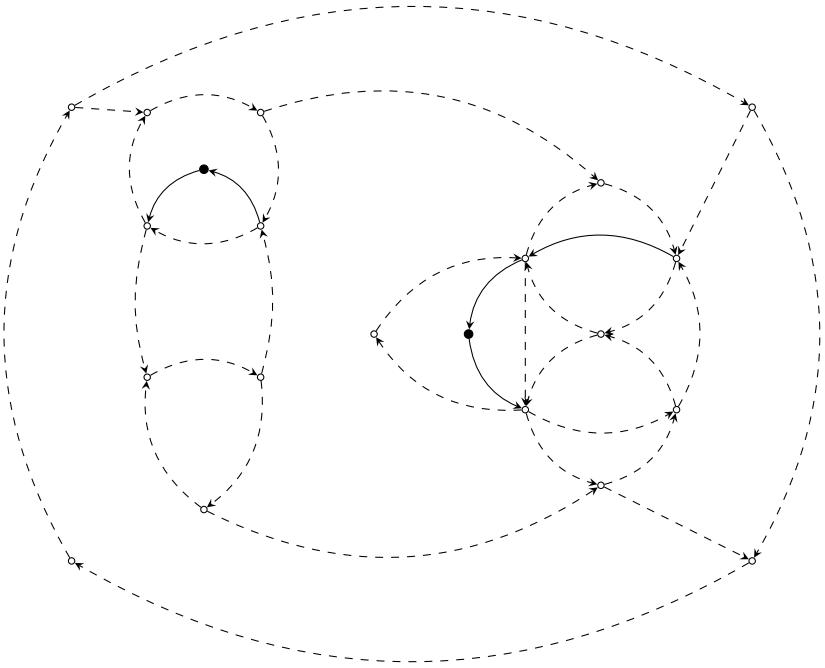


Figure 6 This figure represents the sublayer $(3, 1)$