Efficient Parallel Algorithms for Directed Planar Graphs [*†]

Steve Guattery *Gary L. Miller*

Abstract

We show that computing the strongly connected components and testing reachability in a directed planar graph can be performed in $O(\log n)$ time using n processors by a randomized algorithm. In the same complexity we can also compute such things as ear decompositions. This improves the results of Kao and Klein [KK90] who showed that these problems could be performed in $O(\log^k n)$ time using n processors for some unspecified constant k. In general we give a general paradigm for contracting a directed planar graph to a point and then expanding it back. Using this new paradigm we then "overlay" our application in a fashion similar to tree parallel tree contraction for trees [MR85, MR89].

1 Introduction

Testing if there exists a path from a vertex x to y in a directed graph is known as the reachability problem. Most graph algorithms either implicitly or explicitly solve this problem. For sequential algorithm design the two classic paradigms for solving this problem are BFS and DFS. They only require time at most proportional to the size of the graph. Parallel polylogarithmic time algorithms for the problem now use approximately $O(M(n))$ processors. Ullman and Yannakakis give a probabilistic algorithm which that works in $O(\sqrt{n})$ time using n processors for sparse graphs, [UY90]. This blow-up in the amount of work for parallel algorithms makes work with general directed graphs on fine

*School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213-3890.

†This work was supported in part by NSF grant DCR-8713489.

grain parallel machines virtually impossible. One possible way around this dilemma is to find useful classes of graph for which the problem is efficiently solvable. In pioneering papers Kao and Shannon [KS89] and Kao and Klein [KK90] showed that the reachability problem and many related problem could be solved in polylogarithmic time using only a linear number of processors. Their methods require one to solve each of many related problems by reducing one problem to another. Each reduction introduces more logarithmic factors to the running time. In the end they used $O(\log^5)$ time to solve the planar reachability problem for multiple start vertices.

In this paper we give a general paradigm for contracting planar graphs to a point. We will show that after $O(\log n)$ rounds of contraction an n-node directed planar graph will be reduced to a point. There have been several contraction rules proposed for undirected planar graphs [Phi89, Gaz91] but this is the first set for a directed planar graph. After we present the rules for contraction it will be a relatively simple mater to "overlay" rules necessary to compute reachability and strong connectivity.

Throughout the paper we will assume that the graph $G = (V, E)$ is a directed embedded planar graph. If an an embedding is not given we can construct one in $O(\log n)$ time using $n/\log n$ processors using the work for Gazit [Gaz91] and Ramachandran and Reif, [RR89]. We assume that the embedding is given in some nice combinatorial way such as the cyclic ordering of the edge radiating out of each vertex.

We feel that the class of directed planar graphs are important for at least two reasons. First, the class includes several important classes including tree and series parallel graphs. Second, the flow graph for may structured programming languages without function calls is planar. Our goal is to develop the basic algorithmic foundation for a class of planar graphs so that a theory of planar flow graphs could be based on it.

The extended abstract is divided into 7 sections. In the second is given the main definitions necessary to define a analyze the diplanar contraction algorithm. In the third we give the contraction algorithm for special case of of a planar DAG. In section 4 we prove the main lemma which make it all work. In section 5 we explain how randomization is used to break symmetry. In section 6 we explain how to test reachability for the general case.

2 Preliminaries

Let $G(V, E)$ be a connected embedded planar digraph with faces F. We say that a vertex of G is a **source(sink)** if its indegree(outdegree) is zero. The **alternation number** of a vertex is the number of direction changes as we cyclically examine the arcs radiating from a vertex. Thus, a source or a sink vertex is a vertex with alternation number zero. Observe that the alternation number is always even. Formally, a **source/sink** vertex is a vertex with alternation number zero. A vertex is said to be a **flow** vertex if the alternation number is two. It is a **saddle** vertex if the alternation number is 4 or more. The alternation number of a face can be defined in a similar way. Here we count the number of time the arcs on the boundary of the face change direction as we traverse its boundary. Thus, a **cycle** face has alternation number zero, a **flow** face has alternation number two, and a **saddle** face has an alternation number greater than two. We denote the alternation number of vertex v_i by $\alpha(v_i)$, and the alternation number of face f_i by $\alpha(f_i)$.

Our approach will be based on the following simple but fundamental theorem which we refer to as the **Poincaré index formula**. We show that it follows directly from Euler's formula.

Theorem 2.1 *For every embedded connected planar digraph, the following formula holds:*

$$\sum_{v\in V}(\alpha(v)/2 - 1) + \sum_{f\in F}(\alpha(f)/2 - 1) = -2$$

Proof: We note that if at each vertex we cycle through its incident edges in order according to the embedding, each transition from one edge to the next results in exactly one alternation either for the vertex or for the face for which the two edges lie on the boundary. If we sum the number of alternations over all vertices, we see that the total number of alternations in the graph is equal to the sum of the degrees of all the vertices, which is equal to twice the number of edges in the graph:

$$\sum_{v\in V}\alpha(v) + \sum_{f\in F}\alpha(f) = 2\cdot|E|.$$

Dividing by two and applying Euler's formula, we get

$$\sum_{v\in V}\alpha(v)/2 + \sum_{f\in F}\alpha(f)/2 = |V| + |F| - 2,$$

which gives us

$$\sum_{v\in V}\alpha(v)/2 - |V| + \sum_{f\in F}\alpha(f)/2 - |F| = \sum_{v\in V}(\alpha(v)/2 - 1) + \sum_{f\in F}(\alpha(f)/2 - 1) = -2.$$

□

This formula is important because it tells us a great deal about the structure of a planar digraph embedding. For example, the discussion above about alternation number tells us the following about the contributions of various types of faces and vertices in this formula:

- Sinks, sources, and cycle faces each contribute -1. These are the only elements that make negative contributions; since the sums must come to -2, it is clear that every embedded planar digraph must have at least two such elements. For example, a strongly connected planar digraph cannot have any sinks or sources, so it must have two cycle faces.

- Flow faces and flow vertices contribute 0. There can be an arbitrary number of such elements.
- Saddle vertices and saddle faces will contribute a positive (integer) amount that depends on their alternation number. Since the sum must always be -2, the embedded graph must contain a sink, source, or cycle face for every pair of alternations beyond the first on some saddle.

We will use the formula below to develop invariants and to help us count (for example, we use it to count particular types of edges).

3 Graph Reduction

In this section we introduce a collection of reduction rules and an associated data structure for planar directed acyclic graphs (DAGs). The procedure discussed is a simplification of a general procedure for planar digraphs; however, in the interest of a straightforward presentation, we have restricted it to work in a special case. We show that a constant proportion of the edges are "operable;" thus, the rules could be implemented as an $O(\log n)$ reduction procedure for planar DAGs.

We preprocess the graph such that the following are true of G. These three properties will remain true throughout the algorithm.

1. All vertices are either of type flow, source, or sink.
2. All saddle faces are strictly alternating. That is, the number of arcs counting multiplicity on a saddle face f is $\alpha(f)$.
3. No vertex has both indegree and outdegree of 1. Such vertices are considered to be internal vertices of **topological edges**; such edges are treated as single edges with

respect to the algorithm, though operations on these edges may require the internal vertices to perform operations such as pointer splicing.

It's not hard to see that any planar DAG can be transformed in $O(\log n)$ time so that the first two conditions are true without changing the reachability of the graph. However, DAGs with saddle vertices will not remain DAGs (saddle vertices will be expanded out into cycles), so we don't consider them in the reachability algorithm presented below.

3.1 Reduction Rules

The reduction rules presented below allow us to convert a graph into a smaller graph such that we can recursively solve the problem on which we're working. Once the problem is solved for the reduced graph, we can expand the graph out in reverse order and generate a solution for the original graph. In applying the rules we may modify the connectivity of the graph. Therefore we will keep a data structure of pointers to maintain this information; this data structure is described in the following subsection.

To make the rules simpler to state, we introduce some terminology before we present the rules. The first set of terms is related to flow faces: Let f be a flow face; then the edges on its boundary decompose into two paths, a left and a right (we refer to any edge that is both on the left and the right path as an **internal** edge). There is also a unique **top** and a unique **bottom** vertex on f. Thus the left path starts at the top vertex and in a counter-clockwise fashion goes to the bottom vertex, and the right goes from top to bottom in a clockwise fashion. A **top(bottom)** edge of f is any edge out of(into) the top(bottom) vertex. An edge may be both a top and a bottom for the same face. An edge is simply a top(bottom) if it is the top(bottom) for some flow face.

We associate a data structures with flow faces that will allow us to maintain connectivity information. For each vertex on a flow face that is not top or bottom we have a **cross-**

pointer, pointing from left to right or right to left. Initially each cross-pointer is set to bottom. Intuitively, the connectivity on f as determined by its cross-pointers and boundary edges should be the same as obtained using using arcs and vertices on the boundary of f or those removed from the interior of f by the reduction rules.

Further, the rank of the left(right) vertices will be maintained on each flow face. Each arc will know the two faces common to it. Using concurrent reads, a leader for each face and topological edge, and a ranking on the faces, the vertices can now coordinate their actions. For example, pointers can now be tested in constant time to see if they are forward pointers: simply test if the head and tail are on the same side of the face.

We also mark edges as follows: an edge is **red** if it is the unique edge into a vertex and **blue** if it is the unique edge out. An edge can be both red and blue. Observe that the red edges of a planar DAG form a forest of divergent trees. The set of blue edges is similar, except that the forest will consist of convergent trees. We note that there are two classes of red and blue edges: those that are red(blue) and have no pointer into their head(out of their tail), which we will refer to as **true red(true blue)** edges, and those that have such pointers, which we will refer to as **pointer red(pointer blue)** edges. In the case of topological edges, a true red(true blue) edge will have no pointers into(out of) any internal vertices.

We refer to a source or sink as **strictly changing** if each edge out of the source(into the sink) is adjacent to both a flow face and a saddle face. We denote types of saddle faces by U_i, where the faces of type i are defined to be all faces f such that $\alpha(f)/2 - 1 = i$. For example, U_1 represents the set of smallest saddle faces.

Finally, the rules listed below represent an abstraction of the reduction procedure that might be applied with slight variations to implement a number of different algorithms. The variations would be algorithm-specific actions that would be performed for each rule; in this paper, such actions will be specified with the algorithm description.

We are now ready to list the reduction rules:

[TB Rule] If an edge e is both T and B and it is not common to a saddle face then remove e. If e is topological, any pointers through internal vertices of e must be adjusted by setting any cross-pointer into a vertex internal to e to point to the vertex pointed to by the cross-pointer out of e on its opposite side. The remaining pointers are unchanged. Information on the structure of the face must be updated (e.g., the ordering on the left and right paths of the face must be updated).

[s$\frac{U}{U}$ Rule and t$\frac{U}{U}$ Rule] If an edge e is common to two saddle faces and out of(into) a source(sink) then delete it.

[Degree 1 Rule] If a source or a sink is of degree 1 then remove it and its edge.

[sUs Rule and tUt Rule] If s and s' are sources of degree at most 4 and distance 4 apart on a saddle face U then identify s and s' and remove the multiple edge. A similar rule is applied to sinks.

[s$\frac{U}{F}$t Rule] If a source s of degree at most 4 is on the boundary of a flow face and two saddle faces that lie on either side of the flow face, and one of the edges out of s on the flow face is incident to a sink t, then remove replace the two flow face edges from s with a single edge as in figure ???.

[sU$_1$, tU$_1$, sU$_2$t Rule] Let s and t be a strictly changing degree 2 source and sink, respectively. If s is common to a U_1 saddle remove s and U_1 as in figure ???. A symmetric rule holds for t. If s and t are common to antipodal points of a U_2 saddle then remove s, t, and U_2 as in figure ???.

[True Red(Blue) Edge Contraction Rule] If e is a true red edge out of a source then contract e. A similar rule holds for true blue edges into sinks.

In addition, the following rule can be applied to shift red edges that are not true red edges away from a source to an adjacent vertex (a symmetric rule applies for non-true blue edges at sinks):

[Red(Blue) Edge Flip Rule] If v is a source with a pointer red edge e out, then remove e and replace the highest (with respect ot the vertex ordering along the flow face) pointer related to e with an edge (if there is more than one such edge, only one need be replaced). If the edge is topological, the rule is a bit more complicated. Consider the case for a red topological edge. Then the flip is performed by deleting the path up to the first internal vertex with a pointer into it. That pointer is replaced by an edge. Symmetric rules hold for pointer blue edges into sinks and for such edges that are topological.

Note that this rule does not make progress in the sense that the number of edges in the graph is not reduced when it is applied. However, it does allow us to "clean up" sources and sinks so that other rules may apply. In the algorithm presented below, the number of pointer red(pointer blue) edges out of(into) any vertex is bounded by a constant (the conflict resolution rules are designed such that only external edges at flow vertices will become pointer red or pointer blue edges [For a flow vertex, we define the four edges at the two alternations to be **outer** edges; there will be two outer in-edges and two outer out-edges. The other edges incident to a flow vertex are referred to as **inner** edges]; the reduction rules and the bounded degree of the reduction graph insure that at most a constant number of these red and blue edges will be introduced at sources and sinks). This means we can do the "clean up" of any source(sink) in constant time. We will refer to a source(sink) that has no pointer red(pointer blue) edges as a **clean source(sink)**.

Definition 3.1 *An arc is* **operable** *if one of the rules would remove it.*

Lemma 3.2 [Flow Face Operability] *An edge between two flow faces is operable if it is neither unique-in nor unique-out (i.e., it is marked neither red nor blue).*

Proof: Because the edge is not unique-out, and because there are no saddle vertices in our graphs, there must be an out edge adjacent in the cyclic ordering at its tail vertex. By the definition of a flow face, the vertex at the tail must be the top vertex of one of the flow faces. Therefore the edge is a T edge. A symmetrical argument shows that the edge is also a B edge. Thus, the edge is operable by the TB rule. □

Lemma 3.3 [Red and Blue Edge Count] *In an embedded planar DAG the number of red and blue edges excluding those incident to a degree-1 vertex is less than or equal to* $2/3$ *the number of edges in the graph.*

Proof: As noted above, the red edges (respectively blue edges) form a forest. For the purposes of this proof, we use the term "trimmed red(blue) tree" to refer to a red(blue) tree minus any degree-1 vertices and the edges incident to them. We will start by proving that each trimmed red tree is incident to a number of edges that are either 1) neither red nor blue or 2) are edges into degree-1 vertices that is greater than or equal to the number of edges in the tree (the proof for blue trees is symmetric).

We start by claiming that every leaf of a trimmed red tree must have two edges out; these edges must either be red edges to degree-1 vertices or non-red, non-blue edges. It can't be of degree two, in which case the second edge would have to be an edge out; in that case the vertex would become an internal vertex in a topological edge, contradicting its being a leaf. If the leaf were of degree 1, that would contradict the tree's being trimmed. Thus, each leaf in a trimmed red tree must be of degree 3 or greater. Since the red edge is the unique edge in, there must be two edges out. These edges can't be blue (unique-out); if they are red they must be to degree-1 vertices or we contradict the assumption that this is a leaf of a trimmed red tree. Otherwise, they must be non-red and no-blue. Therefore the claim must hold.

Next we will pair each edge in the tree with an edge out of the tree that is either a non-red, non-blue edge or a red edge incident to a vertex of degree 1. Each edge in the tree must be into either an internal node or a leaf node. We pair each edge into a leaf with one of the edges out of the leaf; this leaves us with one additional edge per leaf. To handle internal nodes, we introduce the following terminology: if an internal node has exactly one tree edge out, we call it a *path node*; otherwise it is a *branch node*. Each path node in a red tree must have an edge out that is either non-red and non-blue or incident to a degree-1 vertex; we pair this edge with the (unique) edge into the path node. The only edges we still have to pair up are those into branch nodes. However, the number of leaves in a tree is easily shown to be greater than the number of branch nodes. Therefore, since we have exactly one red edge into any branch node, we have fewer edges into branch nodes than we have edges left at the leaves. The edges we've associated with each trimmed red tree edge are clearly distinct, and since we have only counted edges out of trimmed red trees, no edge we've counted could be counted for more than one tree. Therefore there is at least one distinct edge of one of our two types for every red edge in the graph that is not incident to a vertex of degree 1.

By a symmetric argument, there is either a distinct non-red, non-blue edge or a distinct blue edge out of a node of degree 1 for every blue edge in the graph that is not incident to a vertex of degree 1. To finish the proof, we observe that each non-red, non-blue edge out of a red tree could also be an edge into a blue tree; thus, in the worst case we might count each of the edges we're interested in twice. In that case the number of edges we've found is at least $1/3$ the number of edges in the graph, from which the lemma follows. □

4 Proof of Main Lemma

Lemma 4.1 [Main Lemma] *In any embedded planar DAG a constant proportion of the edges are operable.*

Proof: See Appendix.

5 Using randomization to Remove Conflict

Definition 5.1 *Two operable edges conflict if:*

- *They are consecutive edges on the boundary of a face.*
- *If the are both marked for deletion and they share a face.*

Observe that each operable edge conflicts with at most 8 other operable edges???

6 A Reachability Algorithm Using Abstract Reduction

In this section we present an algorithm that computes reachability for a special class of embedded planar DAGs using $O(n + m)$ processors in $O(\log n)$ time in the CRCW PRAM model of computation. Specifically, we will work with DAGs that have no saddle vertices; this will allow us to give a simpler presentation. Given an embedded DAG of this type and a set of specified vertices, we want to mark the vertices in the graph that can be reached by a directed path from one of the specified vertices.

We start by assigning one processor to each edge and each vertex in the DAG. We preprocess the DAG as noted in section 3. We note that the preprocessing may add edges, but the number of new edges (which insure that saddle faces are strictly alternating) is at most proportional to the number of edges already in the graph. Note that because we don't have any saddle vertices, we won't add any vertices during the preprocessing.

The algorithm is based on the idea of markers that are propagated through the graph, marking vertices and edges as they move. Each vertex has two flags, one indicating whether a marker has passed through the vertex (referred to as the "marked" flag), the other indicating if the vertex can currently propagate markers to other nodes (referred to as the "propagate" flag). An edge will only have propagate flag and if a progate flag gets set it always sets the marked flag. Initially the set of nodes for which we want to determine reachability have both flags set; no flags are set at any other node or edge.

The algorithm consists of a contraction phase followed by an expansion phase. In the contraction phase we reduce the graph, propagating markers as necessary, until the graph is small enough to deal with in constant time. In the expansion phase we go in reverse order through the series of reductions made during the contraction phase, again propagating markers as necessary. For each vertex and edge we keep sufficient information to support the expansion phase. In particular, we will need an extra pointer at each vertex laying on a flow face. The pointer at a vertex v gives the lowest vertex on the opposite side which can reach v using cross pointers and left and right edges, called the **in-cross pointer**. We modify the reduction rules in the following ways to support these phases:

> **[TB Rule]** In the Contraction Phase, when a topological edge e is removed, we simply pass propagate marks from each vertex v being removed to any vertex that is pointed to by v. We assume that we have passed marks down the vertices of e as the topological path was created. In the Expansion Phase, when we restore a topological edge that was removed by this rule, each vertex v gets a propagate mark from is

in-cross pointer.

[s$\frac{U}{U}$ Rule and t$\frac{U}{U}$ Rule] During the contraction phase, when applying the s$\frac{U}{U}$ rule, if the source has the marker propagation flag set, then set both flags at the vertex at the head of the edge being removed. During the expansion phase, if the vertex at the tail of the edge being restored has the marker propagation flag set, set the "marked" flag at the sink.

[Degree 1 Rule] During the contraction phase, when applying this rule, if the source has the marker propagation flag set, then set both flags at the vertex at the head of the edge being removed. During the expansion phase, if the vertex at the tail of the edge being restored has the marker propagation flag set, set the "marked" flag at the sink.

[sUs Rule and tUt Rule] When applying the sUs rule during the contraction phase, do the following at each source: if the source has the marker propagation flag set, pass the propagation mark to the edges out of this source, then unset the marker propagation flag in the source. Then apply the rule. When applying the tUt rule we must be careful when a sink is the identification of more than one sink. For this reason we only mark the edges into such a sink. We propagate the mark to sink in the expansion phase then the sink consists of single orignal sink.

[s$\frac{U}{F}$t Rule] During the contraction phase, when applying the s$\frac{U}{U}$ rule, if the source has the marker propagation flag set, then set both flags at the vertex at the head of the edge being removed. During the expansion phase, if the vertex at the tail of the edge being restored has the marker propagation flag set, set the "marked" flag at the sink.

[sU_1, tU_1, sU_2t Rule] During the contraction phase, when applying the $s\frac{U}{U}$ rule, if the source has the marker propagation flag set, then set both flags at the vertex at the head of the edge being removed. During the expansion phase, if the vertex at the tail of the edge being restored has the marker propagation flag set, set the "marked" flag at the sink.

[True Red(Blue) Edge Contraction Rule] During the contraction phase, when applying the $s\frac{U}{U}$ rule, if the source has the marker propagation flag set, then set both flags at the vertex at the head of the edge being removed. During the expansion phase, if the vertex at the tail of the edge being restored has the marker propagation flag set, set the "marked" flag at the sink.

Theorem 6.1 *The DAG reachability algorithm will, given a DAG and a specified set of vertices, terminate with each vertex in the graph marked iff it is reachable from at least one of the specified vertices.*

Proof: □

Theorem 6.2 *The DAG reachability algorithm will terminate in $O(\log n)$ time using n processors.*

Proof: The processor count follows immediately from the processor assignment of the algorithm. The proof of the time bound follows by noting that the preprocessing and topological edge cleanup phases take at most time $O(\log n)$, and that the Main Lemma (Lemma 4.1) implies the contraction phase of the algorithm can be completed in $O(\log n)$ time. Since the expansion phase takes the same number of steps as the contraction phase, the theorem follows. □

A Proof of Main Lemma

Lemma A.1 [Main Lemma] *In any embedded planar DAG a constant proportion of the edges are operable.*

Proof: The lemma follows immediately from Lemmas A.2 and A.6 below, which prove the result for two cases that depend on the ratio of sources and sinks to the number of vertices in the graph. □

Lemma A.2 *In any embedded planar DAG where the number of sources and sinks is less than* $n/14$, $1/16$ *of the edges are operable.*

Proof: Assume that the graph has n vertices, of which k are sinks or sources. We first use Lemma 3.3 and the fact that the number of red and blue edges incident to a vertex of degree 1 is bounded by the number of sinks and sources to show that, given the condition in the statement of the lemma, at least $1/4$ of the non-red, non-blue edges in the graph are operable. We then apply the lemma again to show that the number of non-red, non-blue edges is at least $1/4$ the number of edges in the graph.

□

The proof in the case where the number of sources and sinks is large depends on the rules that operate on sources, sinks, and flow faces. We will show that if we can operate on a number of edges proportional to the number of sources and sinks, we can operate on a constant proportion of all edges. To facilitate our counting, we introduce the concept of a source's or sink's kingdom. Letting s-t-count(f) represent the number of distinct sources and sinks that lie on saddle face f, we can make the following calculation: The size of the **kingdom** of a source or a sink v is computed in the following way: If v is not on the boundary of any saddle face, its kingdom has size 0. If v lies on one or more saddle faces, its kingdom size is the sum of the contributions to v of each such saddle face.

The way to calculate the **contribution** of a saddle face f is as follows: There will be some number of edges with v as one of their end points that lie on the boundary of f. For each such edge in turn, we make an undirected traversal of f's boundary. The traversal proceeds until another source or sink is reached, in which case we add half the path length to that source or sink to f's contribution to v, or until we return to v via an edge different from the one we left on, in which case we add in the full path length to the contribution and delete the return edge from the list of edges yet to be processed. Note that some vertices may appear on the face more than once; in that case, when traversing such a vertex we choose the next edge in our traversal consistent with the choice we've made at previous vertices (clockwise or counterclockwise with respect to the embedding). Once all traversals have been made, we subtract 2/s-t-count(f) from the total amount contributed by the traversals.

As a technicality in the counting above, we consider any degree-1 vertex lying on a saddle face to have degree 2, where the single edge into or out of the vertex is actually two parallel edges.

Note that kingdom is defined in such a way that any saddle face f that contributes to one or more kingdoms contributes a total amount equal to $\alpha(f) - 2$.

We call a source or sink with a kingdom of size 2 $1/3$ or greater **uncommon**; a source(sink) that is not uncommon is **common**.

Lemma A.3 *In an embedded planar DAG with a total of k sources and sinks, more than $k/7$ of the sources and sinks are common.*

Proof: Let $V_{s,t}$ be the set of vertices that are either sinks and sources. Let $K(v)$ be the size of the kingdom of v where $v \in V_{s,t}$. Consider the sum of $K(v)$ taken over all $v \in V_{s,t}$. As noted above, each saddle face that has at least one source or sink in its boundary will contribute $\alpha(f) - 2$ to the sum. Each saddle face has alternation number greater than 4, so this quantity is always nonnegative. Thus we have

$$\sum_{v \in V_{s,t}} K(v) \leq \sum_{f \in \mathcal{U}} (\alpha(f) - 2),$$

where $\mathcal{U}$ represents the set of all saddle faces in the graph. We note that for DAGs meeting our invariants (i.e., no saddle vertices),

$$\sum_{v \in V} (\alpha(v)/2 - 1) = \sum_{v \in V_{s,t}} (\alpha(v)/2 - 1) = -1 \cdot |V_{s,t}| = -k$$

and

$$\sum_{f \in F} (\alpha(f)/2 - 1) = \frac{1}{2} \sum_{f \in \mathcal{U}} (\alpha(f) - 2).$$

Combining these with the Poincaré index formula and the inequality above gives us

$$\sum_{v \in V_{s,t}} K(v) \leq \sum_{f \in \mathcal{U}} (\alpha(f) - 2) = 2(|V_{s,t}| - 2) = 2k - 4.$$

Clearly this inequality is violated if 6/7 of the sources and sinks have kingdoms of size 2 1/3 or more, even if the remaining sources and sinks have kingdoms of size 0. □ Note that although this proof is for DAGs, the result holds for planar digraphs in general. Also note that the proof of the lemma implies that any set of sources or sinks with average kingdom size greater than or equal to 2 1/3 must include fewer than 6/7 of the sources and sinks.

Lemma A.4 *Any clean common source or sink can either be associated with at least one operable edge or with a set of sources and sinks with average kingdom size greater than or equal to* 2 1/3.

Proof: We prove this by enumerating the types of clean sources and sinks, and showing that each type either has an operable edge or is uncommon.

- If a source has a true red edge out, we can apply the True Red(Blue) Edge Contraction Rule and contract it. The same holds for sinks with true blue edges. Therefore we only need to worry about cases in which the sources and sinks are not endpoints of red or blue edges.

- If the source or sink is an endpoint of an edge that lies between two flow faces, Lemma 3.2 and the fact that we're only considering sources and sinks with no red or blue edges give us that edge is operable.

- If the source or sink is an endpoint of an edge that lies between two saddle faces, the edge is operable by the $s\frac{U}{U}$ Rule or the $t\frac{U}{U}$ Rule.

In the remaining cases, we can assume that the faces around sources and sinks are strictly changing.

- First consider the case where the source or sink has degree 2. We consider cases based on the distance to the nearest source or sink on the (single) adjacent saddle face:

  - There is no other source or sink on the saddle face. There are two subcases to consider: first, the saddle face has alternation number 4 (i.e., the face has four edges), in which case we can apply the sU_1 or tU_1 Rule, giving us the operable edge(s) we want. Second, the alternation number is higher than 4, in which case the source or sink we are considering has a kingdom of size 4 or greater, and is thus not common.

  - There is a source or sink at distance 1. Consider the case where the vertex we are concerned with is a source (the case for a sink is symmetric). Then the vertex at distance 1 will be a sink. In that case, the $s\frac{U}{F}t$ rule will apply, giving us the desired operable edge. Note that sU_1 or sU_2t rules might also apply, depending on the alternation number of the saddle face and the placement of other sinks on the saddle face.

- There is a source or sink at distance 2. Again, we'll look at the case where the vertex we are concerned with is a source (the case for a sink is symmetric). In this case, there will be a source at distance 2, so we can apply the sUs rule. This gives us the desired operable edge.
- The nearest source or sink is at distance 3. There are two possible cases: First, there are two distinct sources or sinks at the end of the undirected paths out of the vertex in question. In that case, the vertex in question has a kingdom size of at least 2 1/3 and is thus uncommon. Second, there is a single vertex at distance 3 along each path. In that case, the vertex in question has a kingdom of size 2, and is thus common. We must deal with the following cases (for simplicity, we'll assume that the vertex we're looking at is a source):
    * The sink at distance 3 is not strictly changing. Then
    * The sink at distance 3 is strictly changing and has degree 2. In this case we can apply the sU_2t rule to get an operable edge.
    * The sink at distance 3 is strictly changing and has degree 4 or greater. In this case, the sink will be uncommon, but it will either be operable or it will have a large kingdom size. In either case we can include both the source and sink in the set having average kingdom size greater than 2 1/3, effectively transfering some of the kingdom of the sink to the source we are considering and making the source uncommon. To show this, we note that the sink and source each get a contribution of 2 from the saddle face

they share. Since the sink has degree 4 or greater, it must have a second saddle face from which it receives a contribution to its kingdom. It is easy to see that the minimum for this additional contribution is 1/3, in which case the sink is adjacent to two sources. There are two cases to consider: if the sink has degree 4, it is operable and uncommon both, so we can transfer the extra kingdom contribution to the source under consideration. If the sink has degree 6 or greater, then it has a kingdom of size 2 2/3, so we again have sufficient value to transfer 1/3 to the source. In the case that the contribution from one of the sink's additional saddle faces is greater than 1/3, we note that it is then greater than 2/3, so we have a sufficiently large kingdom to support the transfer. Note that we only transfer a kingdom contribution across a face that contributes 2 to each of the kingdoms that lie on it, so there is no problem if we must make multiple contributions.

- Next, consider the case where the source or sink has degree greater than 2 (note that for the source or sink to be strictly changing, the degree must be even, so the condition here is effectively that the degree of the source or sink is four or more). Again, we will assume we are looking at a source and note that the case for a sink is symmetrical. If our source has degree 6 or greater, then it must be uncommon. Therefore we only need to consider the case where the degree of our source is 4. To prove the result for such a source, we consider cases based on the distance to the nearest source or sink on one of the adjacent saddle faces:

  - There are no sources or sinks on either saddle face. Then the kingdom size of the vertex in question is at least 4, and it is uncommon.

- There is a sink at distance 1 on one of the faces. Then we can apply the s t Rule.
- There is a source at distance 2. Then we can apply the sUs Rule to get our operable edge.
- The nearest source or sink is at a distance greater than 2. In this case our source will receive a contribution of at least 2 from each of the saddle faces, so it is clearly uncommon.

□

Lemma A.5 *Each edge associated with a clean common source or sink in the previous lemma is associated with at most a constant number of sources or sinks.*

Proof: This follows from the reduction rules. □

Lemma A.6 *In any embedded planar DAG where the number of sources and sinks is greater than $n/14$, a constant proportion of the edges are operable.*

Proof: The reduction algorithm will keep the number of pointer red edges out of any flow vertex to a constant. We can apply flip rule to clean up any pointer red or blue edges at sources or sinks in constant time. If the graph has no parallel edges, Euler's formula gives us average degree of three times the number of vertices for the rest of the graph. Since at least $1/98$ of vertices are common sources or sinks, and since we have a proportional number of operable edges, it's clear that constant proportion of non-parallel edges is operable. If the graph does have parallel edges, then we note that they must form flow faces. In particular, we can add at most one parallel edge that might be inoperable; the additional edges are all operable. Thus, parallel edges can at most halve the fraction of operable edges. □

References

[Gaz91] H. Gazit. Optimal EREW parallel algorithms for connectivity, ear decomposition and st-numbering of planar graphs. In *Fifth International Parallel Processing Symposium*, May 1991. To appear.

[KK90] Ming-Yang Kao and Philip N. Klein. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. In *Proceedings of the 22th Annual ACM Symposium on Theory of Computing*, Baltimore, May 1990. ACM.

[KS89] Ming-Yang Kao and Gregory E. Shannon. Local reorientation, global order, and planar topology. In *Proceedings of the 21th Annual ACM Symposium on Theory of Computing*, pages 286–296. ACM, May 1989.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–489, Portland, Oregon, October 1985. IEEE.

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, 1989. Vol. 5.

[Phi89] Cynthia Phillips. Parallel graph contraction. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 148–157, Santa Fe, June 1989. ACM.

[RR89] Vijaya Ramachandran and John Reif. An optimal parallel algorithm for graph planarity. In *30th Annual Symposium on Foundations of Computer Science*, pages 282–287, NC, Oct-Nov 1989. IEEE.

[UY90] Jeffery Ullman and Mihalis Yannakakas. High-probability parallel transitive closure algorithms. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, Crete, July 1990. ACM.