A Contraction Procedure for Planar Directed Graphs

Stephen Guattery* Gary L. Miller†

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We show that testing reachability in a planar DAG can be performed in parallel in $O(\log n \log^* n)$ time ($O(\log n)$ time using randomization) using $O(n)$ processors. In general we give a paradigm for contracting a planar DAG to a point and then expanding it back. This paradigm is developed from a property of planar directed graphs we refer to as the Poincaré index formula. Using this new paradigm we then "overlay" our application in a fashion similar to parallel tree contraction [MR85, MR89]. We also discuss some of the changes needed to extend the reduction procedure to work for general planar digraphs. Using the strongly-connected components algorithm of Kao [Kao91] we can compute multiple-source reachability for general planar digraphs in $O(\log^3 n)$ time using $O(n)$ processors. This improves the results of Kao and Klein [KK90] who showed that this problem could be performed in $O(\log^5 n)$ time using $O(n)$ processors. This work represents initial results of an effort to develop efficient algorithms for certain problems encountered in parallel compilation.

1 Introduction

Testing if there exists a path from a vertex x to a vertex y in a directed graph is known as the reachability problem. Many 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, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices together in parallel. 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-grain parallel machines virtually impossible. One possible way around this dilemma is to find useful classes of graphs for which the problem can be solved efficiently. 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 directed acyclic graphs (DAGs) to a point. We will show that after $O(\log n)$ rounds of contraction an n-node directed planar DAG 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 class of directed planar graphs. After we present the rules for contraction it will be a relatively simple matter to "overlay" rules necessary to compute multiple-source reachability.

These results are part of a larger effort to develop a set of reduction rules for arbitrary planar directed graphs (i.e., those with cycles as well as DAGs). The algorithm for the general case is more complicated and is not presented here, though we discuss changes involved in ex-

*This work supported in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No7597, and by National Science Foundation Award CCR-8858087.

†This work supported in part by National Science Foundation grant DCR-8713489.

tending the reduction procedure to the general case. 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 many 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.

In the interest of simplicity we only present the details of the DAG case here. On the other hand, we feel that our algorithm for planar DAGs is interesting in its own right. First, ignoring our algorithm for the general case, we can improve the computation of many-source reachability by a factor of $\log^2 n$ time by simply using the strong connectivity of Kao [Kao91]. Our algorithm for general planar digraphs removes one further $\log n$ factor. Second, it uses new topological techniques, in particular, the Poincaré index formula. This should be of interest in parallel algorithm design for digraphs.

Throughout the paper we will assume that the graph $G = (V, A)$ is a directed embedded planar graph. If an an embedding is not given we can construct one in $O(\log n)$ time using n processors using the work of 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 arcs radiating out of each vertex.

This paper is divided into seven sections. The second gives the main definitions necessary to define and analyze the directed graph contraction algorithm. The third gives the contraction algorithm for special case of of a planar DAG. The theorems in Sections 4 and 5 show that the reduction algorithm for planar DAGs works in a logarithmic number of reduction steps. The sixth section explains how the reduction procedure can be applied to the many-sources reachability problem and calculates the running time. Finally, in Section 7 we discuss work in progress, including some of the steps necessary to extend this result to the case of general planar digraphs.

2 Preliminaries

2.1 Planar Directed Graphs

We will assume that the reader is familiar with basic definitions and results from graph theory that apply to undirected graphs (see, for example, textbooks such as the one by Bondy and Murty [BM76]).

A **directed graph (digraph)** $G(V, A)$ is a set of vertices V and a set of arcs A. Each arc $a \in A$ is an ordered pair drawn from $V \times V$. We say that arc $a = (u, v)$ is directed from u to v; u is the **tail** and v is the **head** of the arc. We say that an arc is **out of** its tail and **into** its head. An arc a is incident to a vertex v if v is the head or the tail of a. The **degree** of a vertex v is the number of arcs incident to it; we represent this number as $degree(v)$. The **in-degree** of a vertex v is the number of arcs that have v as their head; the **out-degree** of v is the number of arcs with v as their tail.

For any directed graph G we can define an undirected graph G' on the same set of vertices in the following way: for each arc (u, v) in G we include an edge (u, v) in G'. We refer to G' as the **underlying graph** of G. In this paper we will distinguish between edges and arcs: edges are undirected and lie in the underlying graph, while arcs are directed. When we refer to arcs in G as edges, we are actually referring to the associated edges in G'.

A **directed path** is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that the v_i's are distinct (with the exception that we might have $v_0 = v_k$) and for all $0 < i \leq k$ we have the arc (v_{i-1}, v_i) in A. A **directed cycle** is a directed path such that $v_0 = v_k$. A digraph that contains no directed cycles is called a **directed acyclic graph (DAG)**.

A **planar directed graph** is a directed graph that can be drawn in the plane in such a way that its arcs intersect only at vertices. A specification of some particular way in which such a graph can be drawn in the plane is called a **planar embedding** of the digraph. In an embedded planar digraph we define **parallel arcs** as two arcs (u_1, v_1) and (u_2, v_2) such that either $u_1 = u_2$ and $v_1 = v_2$ or $u_1 = v_2$ and $v_1 = u_2$, and the arcs are consecutive in the cyclic order at both u_1 and v_1. **Parallel edges** in the underlying graph are edges associated with parallel arcs in the graph.

If the points corresponding to the arcs in an embedded planar digraph are deleted, the plane is divided into a number of connected regions. These regions are called **faces**. The **boundary** of a face is the set of arcs that are adjacent to that face. We denote the set of faces by F. **Euler's formula**, which holds for embedded connected planar graphs, relates the numbers of arcs, vertices, and faces:

$$|V| - |E| + |F| = 2. \tag{1}$$

If the graph also has 3 or more vertices, no self-loops, and no parallel edges, then each face will have at least three edges in its boundary, and it is easy to prove the following inequality:

$$|E| \leq 3 \cdot |V| - 6. \tag{2}$$

The formula corresponding to (1) (with $|A|$ substituted for $|E|$) holds for embedded planar digraphs that have a connected underlying graph since the orientations of

the arcs do not affect the quantities involved. The inequality corresponding to (2) (with $|A|$ substituted for $|E|$) holds for an embedded planar digraph G if G's underlying graph G' is connected and has no self-loops or parallel edges.

2.2 The Poincaré Index Formula

Let $G(V, A)$ be a connected embedded planar digraph with faces F. We say that a vertex of G is a **source(sink)** if its in-degree(out-degree) is zero. The **alternation number** of a vertex is the number of direction changes (i.e., from in to out or vice versa) as we cyclically examine the arcs radiating from a vertex. Observe that the alternation number is always even. Thus, a source or a sink has 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. Vertex alternations are indicated by asterisks in Figure 1.

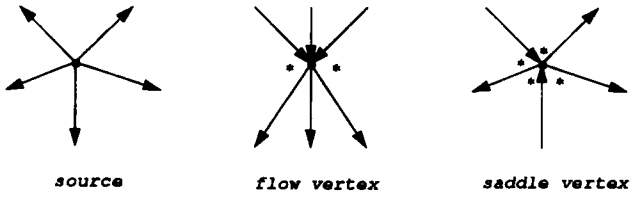

Figure 1

The alternation number of a face can be defined in a similar way. Here we count the number of times 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. Face alternations are indicated by asterisks in Figure 2 below.

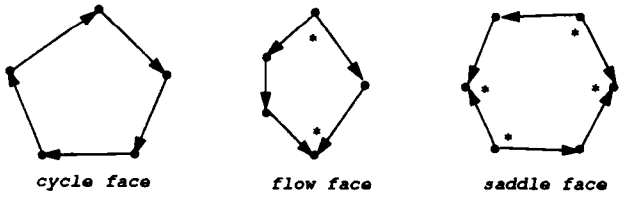

Figure 2

We denote the alternation number of vertex v by $\alpha(v)$, and the alternation number of face f by $\alpha(f)$ (it will be clear from the context whether α refers to a vertex or a face).

A concept related to alternation number is **index**. The index of a vertex v (denoted $index(v)$) is defined as $index(v) = \alpha(v)/2 - 1$. The corresponding definition holds for the index of a face. Once again we do not distinguish between the notation used in these two cases.

Our approach depends on combinatorial arguments based on the following simple but fundamental theorem which we refer to as the **Poincaré index formula**.

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

$$\sum_{v \in V} index(v) + \sum_{f \in F} index(f) = -2.$$

A proof of this result is given in the Technical Report [GM92]. The proof applies the Euler formula along with the observation that if at each vertex we cycle through its incident arcs in order according to the embedding, each transition from one arc to the next results in exactly one alternation either for the vertex or for the face for which the two arcs lie on the boundary (see Figure 3); each alternation is adjacent in this way to exactly one vertex.

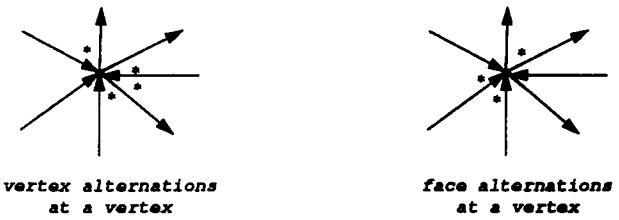

Figure 3

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

- Sinks, sources, and cycle faces each have index -1. These are the only elements that make negative contributions to the sums in the formula; 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 have index 0 and contribute 0 to the sum. There can be an arbitrary number of such elements.

- Saddle vertices and saddle faces have positive (integer) indices that depend on their alternation numbers. 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 each 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 arcs).

2.3 Models of Parallel Computation

The reduction algorithm is specified for the **Parallel Random-Access Machine (PRAM)** model of computation. We discuss the algorithm for this model in the cases where memory accesses are allowed to be concurrent read, concurrent write (CRCW). We also assume the ARBITRARY model for concurrent writes (i.e., an arbitrary one of the values being written to a memory location during a concurrent write will end up in that location).

We have also considered this algorithm in terms of the exclusive read, exclusive write (EREW) model plus unit-time SCANS (see Section 7). This model is based on Blelloch's parallel vector models [Ble90].

3 Graph Reduction

In this section we introduce a collection of reduction rules and an associated data structure for planar DAGs. The reduction rules 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 Sections 4 and 5 we show that at each stage the reduction process removes a constant proportion of the arcs; thus, the rules could be implemented as an $O(\log |A|)$-step reduction procedure for planar DAGs. Inequality 2 in Section 2.1 thus implies that the reduction procedure is $O(\log n)$ (where $n = |V|$ in the original graph). The rules listed below represent an abstraction of the reduction procedure that can be applied with slight variations to implement different algorithms. These variations would be algorithm-specific actions that would be performed for each rule; we will specify such actions in the algorithm description in Section 6.

We will assume that the input to the algorithm is a connected, embedded planar DAG G that has no parallel arcs (and hence no parallel edges). We preprocess the graph such that the following are true of G (these properties will remain true throughout the algorithm):

1. G has only flow faces. This can be accomplished by putting a source in each saddle face, and putting an arc from this source to every vertex that is a local source with respect to the saddle face boundary (Figure 4). It is straightforward to show that the number of edges and hence the number of vertices increases by at most a constant.

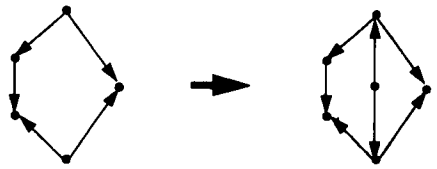

Figure 4

2. No vertex has both in-degree and out-degree of 1 (i.e., there are no degree-2 flow vertices). Such vertices are considered to be internal vertices of **topological arcs**; such arcs are treated as single arcs with respect to the algorithm, though operations on these arcs may require the internal vertices to perform operations such as splicing connectivity pointers. For topological arcs we define the **leader** as the first arc from the original graph in the topological arc (i.e., the arc into the first internal vertex of the topological arc). The rank of the vertices will be maintained on each topological arc.

It's not hard to see that any connected, embedded planar DAG can be transformed in $O(\log n)$ time so that these conditions are true without changing the reachability with respect to the vertices in the original graph.

3.1 Terminology

In order to simplify the presentation of the reduction rules, we first introduce some concepts and terminology.

Let f be a flow face; then the arcs on its boundary decompose into two paths, a left and a right (we refer to any arc that is both on the left and the right path as an **internal** arc). 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 (with respect to the face) goes to the bottom vertex, and the right goes from top to bottom in a clockwise fashion[1]. A **top(bottom)** arc of f is any arc out of(into) the top(bottom) vertex. An arc may be both a top and a bottom arc for the same face. An arc is referred to simply as top(bottom) if it is the top(bottom) arc for some flow face. We will mark top arcs with "T" and bottom arcs with "B."

In applying the rules we may modify the connectivity of the graph. Therefore we associate a data structure

[1] Clockwise and counterclockwise *with respect to a face* can be understood in terms of the dual graph; the clockwise order of edges on the boundary of a face is the same as the order of the corresponding edges in the clockwise cyclic order at the dual vertex corresponding to the face.

with flow faces that will allow us to maintain connectivity information. For each vertex on a flow face that is neither a top or bottom vertex we have a **cross-pointer**, pointing from left to right or right to left. Initially each cross-pointer is set to the bottom vertex. Intuitively, the connectivity on f as determined by its cross-pointers and boundary arcs 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. For each vertex other than top and bottom on a flow face we will also keep the highest and lowest vertex on the opposite side of the face that point to this vertex (initially the high point in will be set to bottom and the low point in will be set to top).

For both the left and right path of each flow face, the top arc will serve as the **leader** of the path (if the top arc is internal it will serve as leader for both sides). Each arc will know the two faces common to it. Using concurrent reads, a leader for each face and topological arc, and the ranking of vertices internal to topological edges, 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. (The coordination actions we will use take constant time in the CRCW model.)

We will refer to saddle vertices by their indices. For example, "saddle vertices with index 1" represents the set of saddle vertices with fewest alternations.

Some reduction rules depend on knowing whether an arc is the unique arc into some vertex or the unique arc out of some vertex. We will refer to such arcs as **unique-in unique-out** arcs. Note that it is possible for an arc to be both unique-in and unique-out. In some cases an arc a might not be unique-in, but at the head of a the next arcs in both the clockwise and counterclockwise cyclic ordering may be out-arcs. In that case we say that a is **locally unique-in**; a symmetric definition holds for **locally unique-out**. Note that we will always use "locally" to imply that there is at least one other edge into(out of) the head(tail), though that edge is not adjacent in the cyclic order.

The existence of topological arcs and the introduction of reachability pointers as described above leads to complications in the application of reduction rules. In particular, we need to distinguish certain unique-in and locally unique-in arcs out of a source. We call such an arc a **clean** if it has the following properties: (1) a has no internal vertices, and (2) for each face f that has a on its boundary, there are no pointers across f into the head of a. Clean unique-out and clean locally unique-out arcs into sinks are defined similarly, with the exception that the second condition prohibits pointers across adjacent faces out of the tail of the arc.

We define the operation of **arc contraction** as follows: the contracted arc is removed from the graph, and the head and tail vertices are combined into a single vertex. The cyclic order of the arcs at this new vertex is the cyclic order at the tail with the arcs at the head vertex inserted (in their original order) where the contracted edge was.

3.2 Reduction Rules

We are now ready to list the reduction rules:

[TB Rule] If an arc a is marked both T and B then remove a. If a is topological, any pointers through internal vertices of a must be adjusted by setting any cross-pointer into a vertex internal to a to point to the vertex pointed to by the cross-pointer out of a on its opposite side. The remaining pointers are unchanged. Information on the structure of the face must be updated (e.g., the left and right leaders), and any new or changed topological arcs must be updated. Pointer updating is shown in Figure 5 (the lighter arrows indicate pointers, the darker ones arcs).

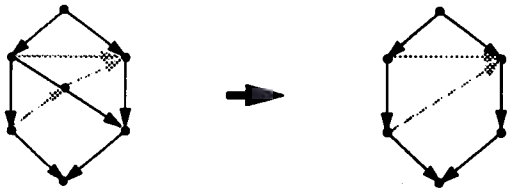

Figure 5

[Degree-1 Rule] If a source or a sink is of degree 1 then remove it and its arc. The leaders on the left and right boundaries of the face are reset if necessary.

[Unique-in(Unique-out) Arc Contraction Rule] If a is a unique-in arc out of a source and a is clean, contract a. The leaders on the left and right boundaries of the face are reset if necessary. The corresponding rule holds for unique-out arcs into sinks.

[Adjacent Degree-2 Sources and Sinks Rule] If a degree-2 source and a degree-2 sink incident to clean arcs are in the configuration shown in Figure 6, remove the source and sink and their arcs as shown (ins this and subsequent figures straight arrows represent arcs and curved arrow represent segments of face boundaries that may be longer than a single arc).

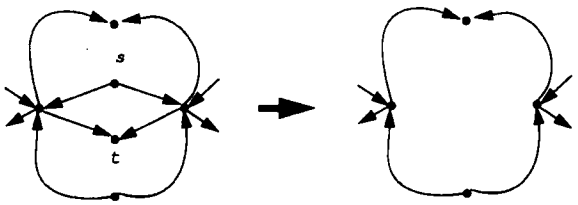

Figure 6

[Source-Sink-Source (s-t-s)/Sink-Source-Sink (t-s-t) Rule] Let s be a degree-2 or degree-3 source having only clean locally unique-in arcs out. If at two of the saddle vertices adjacent to s there are clean locally unique-out arcs out of distinct sinks adjacent in the cyclic order to the arcs from s, take the following actions:

- If s has degree 2, remove the source and its arcs, and combine the two sinks into a single sink (see Figure 7a - since all faces are flow faces, each sink will be at the bottom of one of the two faces on the boundaries of which s lies).
- If s has degree 3, remove the arc out of s common to the two faces on the boundaries of which the two sinks lie, then combine the two sinks into a single sink (Figures 7b and 7c).

A corresponding rule applies for sinks and adjacent sources. If a large number of vertices are combined into a single vertex, a processor must be selected to represent that vertex. Although this could take time $O(\log n)$, this computation can be done in parallel with the rest of the algorithm without affecting the running time.

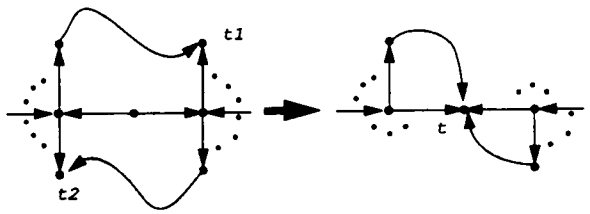

Figure 7a

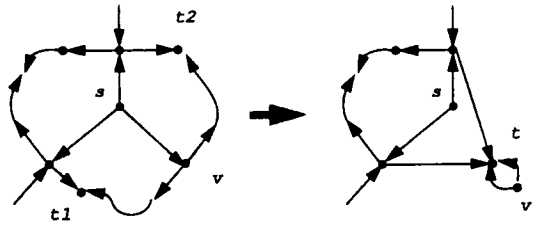

Figure 7b

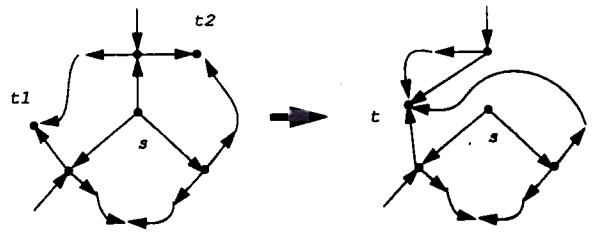

Figure 7c

[Consecutive Rule] Let s be a source with a clean locally unique-in arc out. If at the head of the locally unique-in arc there are clean locally unique-out arcs into sinks adjacent in the cyclic order in both the clockwise and counterclockwise directions, do the following: remove the locally unique-in arc from the source, combine the two sinks into a single sink, and combine the two locally unique-out arcs into a single arc (see Figure 8). A corresponding rule applies for a sink and adjacent sources.

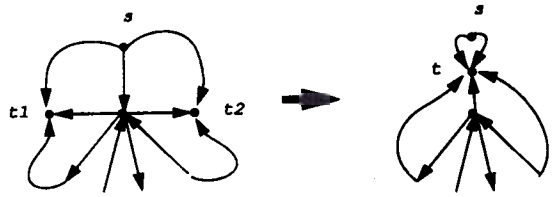

Figure 8

[Index 1 Saddle Rule] If a source has a clean arc to a saddle vertex of index 1 and if the only other arc into the saddle is a clean arc from another source, then contract one or both of the in-arcs (see Figure 9a). A corresponding rule holds if there are exactly two clean out-arcs to sinks (Fig. 9b). As for the s-t-s and t-s-t Rules, if a large number of vertices are combined into a single high-degree vertex, a processor must be selected to represent that vertex.

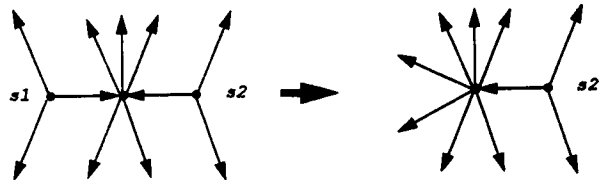

Figure 9a

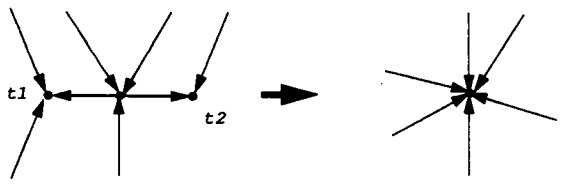

Figure 9b

In the CRCW model it is easy to determine in constant time if the conditions for rule application are met. These conditions can be checked locally in the graph. Ignoring the time to do conflict resolution for now, rule applications can be done in constant time.

3.3 Cleaning Up the Graph

Many of the rules above require that the arcs involved be clean. These arcs will not necessarily be clean, however. Therefore we introduce a parallel algorithm for cleaning up arcs out of sources(into sinks) that runs in constant time in the CRCW model. The cleanup algorithm will be run as a subroutine in the reduction algorithm. It will also be run with respect to an invariant relative to the particular problem to which the reduction algorithm is applied (e.g., in the case of many-source reachability the essence of the invariant is that the vertices in the current graph that are reachable from one of the starting vertices are either marked as reachable or have a path consisting of pointers and edges from some vertex that is marked). Applying cleanup with respect to the invariant will add computation to the cleanup algorithm; in general, we try to choose an invariant in such a way that it doesn't increase the asymptotic running time of the basic cleanup algorithm.

We do not clean up all sources and sinks. To insure that cleanup doesn't take too long, only sources and sinks of degree less than or equal to a constant d (to be specified later) and incident only to unique-in or locally unique-in(unique-out or locally unique-out) arcs will be cleaned up (note that such sources and sinks have no parallel arcs out or in). Not all of these conditions are necessary to insure limited running time; some are related to the proofs in Section 4.

Details of the cleanup algorithm are presented in the Technical Report.

3.4 Overview of the General Reduction Process

The general algorithm for reducing an embedded planar DAG can now be stated:

1. Preprocess the graph to make it consistent with our invariants.
2. Main Reduction Loop: While there are arcs left in the graph, repeat the following sequence of steps:
   - Perform any application-specific actions.
   - Clean up the current graph.
   - Apply the reduction rules in the order they're listed in Section 3.2. Application-specific processing may be required as each rule is applied.
3. Perform any application-specific processing needed prior to the expansion phase.
4. Reconstruct the graph by reversing the steps in the reduction process (note that this requires that we have stored, in order, all changes made during the reduction process).

4 Operability Lemmas

In the next two sections we prove that the reduction procedure given above works in $O(\log n \log^* n)$ time using $O(n)$ processors. We start by showing that at each step of the algorithm a constant proportion of the arcs are candidates for removal.

Definition 4.1 *An arc is* **operable** *if one of the rules would remove it.*

Lemma 4.2 [Operability Lemma] *In any connected, embedded planar DAG consistent with our invariants a constant proportion of the arcs are operable.*

Proof: The lemma follows immediately from Lemmas 4.3 and 4.5 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 4.3 *In any connected, embedded planar DAG consistent with our invariants and in which the number of sources and sinks is less than $n/14$, $1/16$ of the arcs are operable.*

In the interest of brevity, we will only outline the proofs of this lemma and Lemma 4.5. Full proofs are given in the Technical Report. To prove this lemma we show that in graphs meeting the conditions stated there are many arcs that either are operable by the Degree-1 Rule, or are operable by the TB Rule. The following lemma is a useful tool in the latter case:

Lemma 4.4 [Flow Face Operability] *An arc between two flow faces is operable if it is neither unique-in, locally unique-in, unique-out, nor locally unique-out.*

Proof: Because the arc is neither globally nor locally unique-out, there must be an adjacent out-arc in the cyclic ordering at its tail vertex, which thus must be the top vertex of one of the flow faces. Therefore the arc is a T arc. A symmetrical argument shows that the arc is also a B arc. Thus, the arc is operable by the TB rule. □

The proof of Lemma 4.3 proceeds by showing that there are many arcs that either unique-in or unique-out and incident to degree-1 sources and sinks, or that are neither unique-in nor unique-out. This follows from the fact that our graphs have no degree-2 flow vertices and that the graph induced by the unique-in and unique-out arcs is a forest. This isn't quite enough to prove the lemma, however; we must then show that most of the non-unique-in and non-unique-out arcs are neither locally unique-in nor locally unique-out. This follows from the Poincaré Index Formula and the conditions of the lemma.

We use a counting argument to prove the second half of Lemma 4.2:

Lemma 4.5 *In any embedded planar DAG consistent with our invariants in which the number of sources and sinks is greater than $n/14$, a constant proportion of the arcs are operable.*

As above, see the Technical Report for a full proof. An outline of the proof follows: First we show that a high degree source or sink v (i.e., $degree(v) > d = 1176$) either is incident to a TB arc or is uncommon in the sense that the number of such sources and sinks is less than a constant proportion of the number of sources and sinks in the graph given the conditions in the lemma statement. Next we show that at least $1/7$ of the sources and sinks with degree $\leq d$ are incident to an operable arc. Such vertices will be processed in the cleanup phase, so any source(sink) incident to an arc that is not locally unique-in (locally unique-out) has an operable arc out(in). In the remaining cases all incident arcs are locally unique-in or locally unique-out; by a counting argument based on the Poincaré Index Formula we show that at least $1/7$ must be incident to an operable arc. The lemma follows from there.

5 Conflict Resolution

In the previous section we showed that in any embedded connected planar DAG meeting certain invariants a constant proportion of the arcs are operable. However, applying the reduction rules to these operable arcs leads to two types of conflicts we must deal with. The first sort of conflict arises when we try to apply a single rule to all arcs that are operable by that rule. Doing so leads to cases in which either our invariants aren't maintained or in which the rule applications cannot be completed in the time available. An example in which invariants aren't maintained include removing both B arcs from a flow face when these arcs are the only arcs into the bottom vertex; an example of a potential time problem is where removal of many TB arcs causes arbitrarily many pointers to be spliced together. The second conflict arises when applications of a particular rule make arcs that were formerly operable by another rule inoperable. This conflict affects our counting argument aimed at showing a constant proportion of the arcs are removed in each pass through the main loop.

5.1 Conflict Resolution for a Single Rule

We will deal with conflicts between applications of a single rule by building a **conflict graph** that relates the conflicting arcs[2]. The graph consists of a vertex for each arc operable by the rule in question, and an edge between each pair of vertices if the removal of the corresponding arcs would cause a problem with respect to our invariants or the running time of the removal step. The edges are undirected because the conflicts are symmetrical here.

The conflict graphs for our rules with one exception have bounded degree (see the Technical Report for details); they are not necessarily planar, however. They are easily constructed in constant time in the CRCW model.

It is obvious that a maximal independent set (MIS) of vertices from a conflict graph represents a set of vertices that can be removed in parallel without problems; it is also obvious that a MIS in a bounded-degree graph contains a constant proportion of the vertices. Therefore we can use the techniques developed by Goldberg, Plotkin, and Shannon [GPS87] to resolve conflicts in $O(\log^* n)$ time. Introducing randomness into the model of computation allows us to find an independent set that includes a constant proportion of the vertices in the conflict graph in constant time.

5.2 Conflict Resolution Between Rules

We deal with the second type of conflict by proving the following lemma:

Lemma 5.1 *In applying any reduction rule in the order specified in the algorithm description at a particular source or sink, the number of arcs operable by subsequent*

[2]The case for the TB Rule is actually somewhat more complicated. We divide the arcs operable by the TB Rule into three classes and build a different conflict graph for each class in turn. See the Technical Report for details.

rules (excluding those removed by this rule application) is reduced by at most a constant number.

The proof is by examination of all cases.

5.3 Proof of Main Lemma

The only step left in proving that the reduction algorithm runs in a logarithmic number of iterations of the main loop is to show that applying the conflict resolution scheme above will allow the reduction algorithm to remove a constant proportion of the operable arcs in each pass through the main loop.

Lemma 5.2 [Main Lemma] *For any embedded connected planar DAG consistent with our invariants, the generalized reduction algorithm will work in $O(\log n)$ iterations of the main loop using $O(n)$ processors.*

Proof: The result follows from Lemma 5.3 below and Lemma 4.3 if the number of sources and sinks is less than $n/14$. It follows from Lemma 5.4 below and Lemma 4.5 otherwise. □

Lemma 5.3 *When applied to any embedded connected planar DAG consistent with our invariants, the generalized reduction algorithm will remove a constant proportion of the TB arcs on each pass through the main loop.*

The proof of this lemma is based on showing that for each TB arc removed at most a constant number of other TB arcs are not removed because of conflicts. This implies that a constant proportion of the TB arcs are removed.

Lemma 5.4 *When applied to any embedded connected planar DAG consistent with our invariants, the generalized reduction algorithm will remove a constant proportion of the operable arcs.*

Proof: For each arc removed, consider the number of operable arcs with which it conflicts. By Lemma 5.1 and the fact that the conflict graph for each rule is bounded, this number is bounded by a constant. Since every operable arc is either removed or is subject to a conflict with an arc that is removed, at least a constant proportion of the operable arcs are removed. □

6 Applications

In this subsection we present an application that uses the abstract reduction procedure presented above. We also present the running time and number of processors needed to run this application.

6.1 Planar DAG Many-Source Reachability

The abstract reduction procedure can be used to solve the many-source reachability problem for planar DAGs. The problem can be stated as follows: given a planar DAG and a set of vertices in that DAG as the input, compute the set of vertices that are reachable via directed paths from the input set of vertices. To do this, we must come up with a set of application-specific actions to take at various points in the reduction algorithm plus an invariant that will allow us to prove that the result is correctly computed.

We introduce two flags at each vertex: a flag indicating whether or not the vertex has been marked as reachable from one of the initial vertices, and an "active mark" flag that we will use to determine where to propagate marks during the reduction phase. The algorithm starts with the input set of vertices having both their "active mark" and "reachable" flags set. Marks are propagated as follows:

- At the start of each iteration of the main loop, each vertex reachable either via a connectivity pointer or a directed arc from a vertex with an active mark sets both of its flags. Any with an active mark unsets its "active mark" flag.
- At the start of each cleanup phase, each active mark at an internal vertex of a topological arc out of a source of degree less than or equal to the degree limit d is propagated across its crosspointers a number of times equal to the degree of the source. Then all active marks at such sources and at vertices internal to topological arcs out of such sources are propagated to the heads of the arcs out of the source. A symmetrical procedure is done at sinks.
- When applying each reduction rule, one or two propagation steps are done for the arcs removed. See the Technical Report for details.

In addition, sinks are never marked during the reduction phase.

Between the contraction and expansion phases, any topological arc removed during contraction that has a mark at the tail or an internal vertex propagates the mark down the arc. This can be done using list ranking techniques.

During the expansion phase, as topological arcs are added back to the graph their internal vertices may need to be marked. This involves checking the lowest crosspointer that can reach each internal vertex to see if its tail vertex is marked. Also during expansion any vertices that became sinks or were merged with sinks are marked as necessary as they revert from being a sink.

Sinks are marked at the end of the expansion phase if they have a marked neighbor.

To prove that this process correctly marks the reachable vertices we use the following invariants, one for the contraction phase and one for the expansion phase. The term **active set** refers to vertices in the current graph that are not sources, sinks, or vertices that represent the combination of two or more vertices from the original graph. The contraction invariant is as follows:

Lemma 6.1 *During the contraction phase a vertex v in the active set that is reachable in the original graph from one of the input set vertices if and only if it is either marked as reachable or there exists an active mark at some vertex u in the active set and a directed path of arcs and crosspointers from u to v that includes only vertices from the active set.*

The expansion invariant is as follows:

Lemma 6.2 *During the expansion phase all vertices in the active set are correctly marked.*

These invariants are sufficient to prove that at the completion of the algorithm the graph is correctly marked. See the Technical Report for details.

6.2 Running Time and Processor Count

The running time is determined by observing that the main loop is executed $O(\log n)$ times in the reduction and expansion phases. The running time of the main loop is dominated by the $O(\log^* n)$ time it can take to resolve conflicts for some of the reduction rules. Preprocessing time is dominated by the time for the main loop, so the running time is $O(\log n \log^* n)$ (this can be reduced to $O(\log n)$ through the use of randomization as noted above). The algorithm can be run using one processor per face, vertex, and arc, which is linear in the size of the input graph. When combined with Kao's strongly connected components algorithm [Kao91] the running time becomes $O(\log^3 n)$.

7 Work in Progress

We are currently working on extensions to other models and to planar digraphs that include cycles.

7.1 Extensions to Other Models

We are in the process of writing up an extension of this work to the exclusive read, exclusive write (EREW) plus unit-time SCANS PRAM model. We expect that the reduction algorithm will run in time $O(\log n)$ using randomization and a number of processors linear in the graph size. Making conflict resolution deterministic by using the techniques of Goldberg, Plotkin, and Shannon will add an additional factor of $\log^* n$ to the time.

7.2 Reducing Planar Digraphs with Cycles

The techniques above can be expanded to work with planar graphs that have cycles. This is particularly useful in that we can then compute strongly connected components, and thus we can compute many-source reachability for any planar digraph (by first computing strongly connected components and then contracting them, then computing many-source reachability, then expanding back out the strongly connected components).

The reduction algorithm for the cyclic case is more complicated, as are the proofs of its correctness. We summarize some of the differences below:

- We must introduce two new rules (an arc contraction rule and an arc removal rule) for cycle faces. The invariant changes to allow cycle faces as well as flow faces.
- In addition to crosspointers on flow faces we must keep backpointers to the highest point reachable on the same side of the face.
- Cleanup is more complex because of the backpointers. We must now clean up two levels of arcs from sources or sinks. In addition, we must spend $O(\log n)$ time determining the connectivity implied by the backpointers during cleanup.
- The operability proofs must be modified to take into account the existence of cycle faces in the graph.

Acknowledgements We would like to thank Ming-Yang Kao, John Reif, and Doug Tygar for their useful discussions, comments, and suggestions.

References

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT-Press, Cambridge MA, 1990.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* North-Holland, New York, 1976.

[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.

[GM92] S. Guattery and G. L. Miller. A parallel contraction procedure for planar dags. Technical Report CMU-CS-92-140, Carnegie Mellon University, 1992. To Appear.

[GPS87] Andrew Goldberg, Serge A. Plotkin, and Gregory Shannon. Parallel symmetry-breaking in sparse graphs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, New York, May 1987. ACM.

[Kao91] Ming-Yang Kao. Linear-processor nc algorithms for planar directed graphs i: Strongly connected components. *Accepted to SIAM Journal on Computing*, 1991.

[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, Greenwich, Connecticut, 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.