# Applications of Graph-Theoretical Properties in Algorithms

Stephen Guattery
September 5, 1995
CMU-CS-95-187

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy*

**Thesis Committee:**
Gary Miller, Chair
Guy Blelloch
Doug Tygar
John Reif, Duke University



This work was supported in part by NSF Grant CCR-9505472.

This work was also supported in part by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193. In addition, IBM, Motorola, and the NSF/Presidential Young Investigator Award under Grant NoCCR-8858087, TRW, and the U.S. Postal Service gave their support.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government, its agencies, or other funders.

Keywords: Algorithms, Graph Theory, Graph Algorithms, Spectral Methods, Numerical Linear Algebra, Eigenvalues, Planar Graphs, Parallel Algorithms

Carnegie Mellon

School of Computer Science

**DOCTORAL THESIS**
**in the field of**
**Computer Science**

# *Applications of Graph-Theoretical Properties in Algorithms*

**STEPHEN GUATTERY**

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

**ACCEPTED:**

| | |
|---|---|
| THESIS COMMITTEE CHAIR | 9/15/95 DATE |
| DEPARTMENT HEAD | 9/28/95 DATE |

**APPROVED:**

| | |
|---|---|
| DEAN | 9-29-95 DATE |

**Abstract**

Graphs are an important means of representing problems of interest to computer science; the relationship between graphs and algorithms is a close one. Many theorems in graph theory have algorithmic proofs in nature; conversely, many algorithms depend on graph-theoretic properties. In this thesis, I introduce and apply graph properties in the analysis of one type of graph algorithm and the design of another.

First, I examine the structure of the eigenvectors of the Laplacian matrix representation of a graph to analyze some commonly-used spectral algorithms for finding separators, an important step in many graph algorithms. There is little previous analysis of the quality of the separators produced by this technique; instead it is usually claimed that spectral methods "work well in practice." I present an initial analysis, considering two popular spectral separator algorithms plus a generalization of such algorithms, and provide counterexamples showing these algorithms perform poorly on certain graphs similar to those arising in practice.

Second, I examine the consequences of the Poincaré index formula for planar directed graphs, Application of the formula leads to a method for reducing a planar DAG to a constant size and then expanding it back. This method can be used to implement an algorithm for testing reachability in a planar DAG in parallel on a CRCW PRAM in $O(\log n \log^* n)$ time ($O(\log n)$ time using randomization) using $O(n)$ processors. In conjunction with Kao's strongly-connected components algorithm, multiple-source reachability for planar digraphs can be computed in $O(\log^3 n)$ time using $O(n)$ processors. This improves the previous algorithm of Kao and Klein, which solved this problem in $O(\log^5 n)$ time using $O(n)$ processors.

# Acknowledgements

Special thanks are due Gary Miller, my advisor and thesis committee chair. Although I use the pronoun "I" throughout this thesis, this work was joint work with Gary and shows his influence and ideas. Gary has all the attributes of a great advisor: deep knowledge of the field, enthusiasm, and an interest in new problems. He has tremendous intuition about mathematical and algorithmic problems and a way of distilling them to their essence. Quite often I've gone to talk to Gary when I was stuck, and returned with a clear idea of directions to pursue. Gary has never hesitated to give me a push when I got bogged down, or when I was ready to abandon some problem because it looked too hard. He always supported my work, and provided good advice on career directions.

Doug Tygar also deserves special thanks. He was my advisor in my early days at CMU; he taught me many things that helped build a foundation for later work. Doug has both great recall and and a great library. I don't recall ever asking him a question that he didn't either answer or find a reference with the answer. He co-advised me after I started working with Gary, providing financial support and good advice on practical issues such as schedules, preparing documents for job search, etc.

I'd also like to thank the other members of my committee, Guy Blelloch and John Reif, for their careful reading and comments on my thesis.

Ming Kao deserves thanks for reading and commenting on Chapter 3. I'd also like to thank Dafna Talmor for reading parts of the thesis and providing valuable comments about presentation and readability.

There are many others who should be mentioned; I'm sorry that space and memory limitations prevent this.

Parts of this work have appeared in the Sixth Annual ACM/SIAM Symposium on Discrete Algorithms (SODA '95) ([GM95b], [GM94]) and the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '92) ([GM92], [GM95a]).

# Chapter 1

# Algorithms and Graph Properties

## 1.1 Introduction

Graphs are a fundamental way of representing many interesting computer science problems. Some important examples include:

- circuit layout;
- compiler optimization;
- the structure of finite element meshes in numerical computation;
- the representation of constraints in scheduling and other optimization problems; and
- network representation.

Graphs and algorithms are closely related. Many theorems in graph theory have algorithmic proofs. Conversely, many algorithms depend on graph-theoretic properties. For example, graph theory proves the existence of a particular graph structure; the algorithm then works on the structure to get the desired result. Examples include the max-flow min-cut theorem for network flows, the use of augmenting paths in matching theory, and the detection of articulation points in biconnected components algorithms (these are described in basic algorithms texts such as [AHU74] or [CLR90]). In other cases, the properties are combinatorial; they are used to bound the running time of an algorithm (e.g., planar graphs have bounded average degree) or show that a certain property does or does not exist (e.g., the use of interlacing bridges in the planar graph embedding algorithm) (see, e.g., basic graph theory texts such as [BM76]).

In this thesis, I introduce and apply graph properties in the analysis of one type of graph algorithm and the design of another type. The first property has to do with Laplacian eigenvector structure and is covered in the Section 1.2; the second is a topological property of embedded planar directed graphs and is covered in Section 1.3.

## 1.2 Properties of Laplacian Eigenvectors

I use a set of properties of the eigenvectors of a particular matrix representation – the Laplacian – of a connected graph, to analyze the performance of spectral separator algorithms on various classes of graphs. In particular, I prove there are classes of graphs (similar to graphs arising in practice) for which three commonly-used spectral separator algorithms perform poorly.

To understand how these properties are used, one needs to understand the meaning of "spectral separator algorithms".

A separator is a set of edges or vertices that, if removed from a graph, breaks the graph into two (not necessarily connected) components. I focus on edge separators in this thesis, and further references to separators in this chapter are to edge separators. Good separators are **small** and break the graph into two pieces of **comparable size**. Small separators are important because many algorithms do work proportional to the separator size. Breaking the graph into pieces of comparable size is important in recursive algorithm design to limit recursion depth.

Separators have many useful applications in VLSI layout, testing, and verification; assignment of tasks to processors in parallel computation; and design and analysis of divide-and-conquer algorithms. However, separators are difficult to compute: finding optimal separators for useful definitions of "small" and "comparable size" is typically NP-hard (see, e.g., [LR88] for details). Thus practical separator algorithms are heuristic in nature.

Spectral methods use the eigenvalues and eigenvectors of a matrix representation of a graph. These methods have a long history, and are used to solve many problems other than finding graph separators. Various matrix representations of graphs are used; in this thesis I focus on the application of spectral techniques to finding separators, and more particularly on spectral techniques applied to the Laplacian of a graph. The algorithms I analyze are widely used in practice because the needed software is available and often fairly fast.

The **Laplacian** $B$ of a graph $G$ on $n$ vertices is the $n \times n$ matrix with the $i^{\text{th}}$ diagonal entry equal to the degree of vertex $i$, and entry $(i, j)$ (for $i \neq j$) equal to $-1$ if edge $(v_i, v_j)$ exists and 0 otherwise. Laplacians have a number of useful properties, including:

- Laplacians are symmetric.
- Laplacians are positive semidefinite.
- Zero is simple eigenvalue for the Laplacian of a connected graph.
- All row sums (and, by symmetry, all column sums) are zero since the diagonal entry is the vertex degree and there is a minus one in the row for each incident edge; hence, the vector with all entries equal to 1 (denoted as $\vec{1}$) is the eigenvector for the eigenvalue 0.

The following identity also holds:

$$\mathbf{x}^T B \mathbf{x} = \sum_{(v_i, v_j) \in E(G)} (x_i - x_j)^2. \tag{1.1}$$

This last fact provides some intuition for why spectral techniques are used. Consider the set of vectors with all entries either 1 or $-1$. Such vectors can represent partitions: vertices with value 1

are on one side of the partition, vertices with value $-1$ are on the other. When a partition vector is plugged into Equation 1.1, edges between a pair of vertices that both lie on one side of the partition contribute 0 to the sum. Edges across the partition contribute 4. Thus Equation 1.1 evaluates to 4 times the number of edges across the partition.

Now I can formally state a particular separator problem, the problem of finding a minimum bisection. A bisection is a separator dividing the graph into two equal-size sets of vertices. The bisection problem can be stated as follows:

Minimize $\mathbf{x}^T B \mathbf{x}$ subject to:

$$x_i \in \{+1, -1\} \tag{1.2}$$
$$\mathbf{x}^T \vec{1} = 0 \tag{1.3}$$

Constraint 1.2 specifies that $\mathbf{x}$ is a partition vector; Constraint 1.3 forces the number of vertices on each side of the partition to be equal.

Solving this problem exactly is NP-complete [PSL90], and there are no known algorithms that provide an optimal solution in polynomial time. Thus, to get a (very likely suboptimal) separator in a reasonable amount of time, heuristic techniques are applied. One possibility is to relax one of the constraints to get an easier problem to solve; the solution of this problem can be converted into a bisection.

Consider relaxing Constraint 1.2. Then, the solution can be a real vector. However, one must avoid the problem that scaling the vector by an amount between zero and one reduces the objective function. Thus, a normalization condition is added to get the following problem statement:

Minimize $\mathbf{x}^T B \mathbf{x}$ subject to:

$$\mathbf{x}^T \mathbf{x} = 1$$
$$\mathbf{x}^T \vec{1} = 0$$

The solution to this problem is a vector of considerable interest in this paper: it is the eigenvector corresponding to the second smallest eigenvector of $B$ (often called the second smallest eigenvector of $B$), which I denote by $\mathbf{u}_2$. Note that each vertex has a corresponding entry in $\mathbf{u}_2$ (or in any vector); I will sometimes to this as the vertex's value.

How does one get a partition vector from this solution? A common way is to partition the vertices of the graph according to the values of their corresponding entries in $\mathbf{u}_2$ [PSL90, HK92]. All vertices with a value greater or equal to the median entry of $\mathbf{u}_2$ go on one side of the partition; all vertices with value less than the median go on the other. This is the first of three spectral algorithms considered in Chapter 2.

While this algorithm works well for some graphs, there are fairly simple graphs for which it does quite poorly. I show a class of bounded-degree planar graphs with constant-size separators for which the above spectral bisection algorithm gives $\Theta(n)$-size separators: **roach graphs**. The roach graph looks like a ladder with the top $2/3$ of its rungs kicked out (see Figure 1.2); a straightforward spectral bisection algorithm cuts the remaining rungs, whereas the optimal bisection is made by cutting across the ladder above the remaining rungs. (I refer to this graph as the "roach graph" because its outline looks roughly like the body of a cockroach with two long antennae.) It is obvious

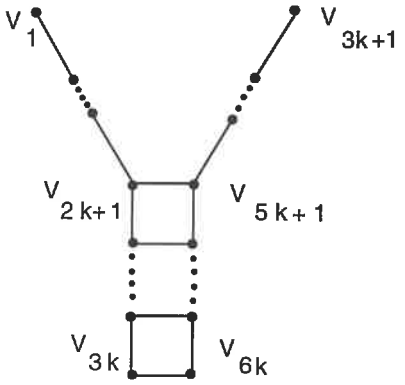



Figure 1.1: The Roach Graph

that the roach graph has a bisection of size 2 (cut the antennae off at the appropriate places); I show that the spectral bisection algorithm cuts through the edges across the body.

To prove this, I show some properties of graphs with particular **graph automorphisms**. A graph automorphism is an edge-preserving permutation $\phi$ on the graph vertices: if $(v_i, v_j)$ is an edge, then $(v_{\phi(i)}, v_{\phi(j)})$ must also be an edge. The roach graph has an automorphism mapping one side of the roach to the other. I define **odd** and **even** vectors with respect to an automorphism $\phi$: Let the graph have $n$ vertices. An even vector $\mathbf{x}$ with respect to $\phi$ has $x_i = x_{\phi(i)}$ for all $i$ in the range $1 \leq i \leq n$; an **odd** vector $\mathbf{y}$ with respect to $\phi$ has $y_i = -y_{\phi(i)}$ for all $i$.

Intuitively, it is easy to see that if $\mathbf{u}_2$ for the roach graph is an odd vector, then the spectral bisection algorithm cuts all the edges across the body: In that case, there are an equal number of positive and negative entries, and the positive entries go on one side of the partition, the negative entries on the other. Since endpoints of the edges across the body are images of each other under $\phi$, they lie in different parts of the partition, and all these edges are cut (this explanation is extremely simplified; Chapter 2 contains the technical details).

There are easy ways to modify the spectral bisection algorithm to give good cuts for the roach graph. These modifications produce separators that do not necessarily divide the vertices into two equal sets. How should one measure the quality of such separators?

Any quality metric for separators must balance the notions of "small separator" against "vertex sets of comparable size". Intuitively, one would like the ratio of the number of edges cut to the size of the smaller set of vertices to be as small as possible (the size of the cut can be thought of as the boundary size of the vertex set; the number of vertices in the smaller set can be thought of as its volume). This notion is formalized in the definition of the **cut quotient**. Let $S$ be an edge separator that divides $G$ into the vertex sets $G_1$ and $G_2$. Then the cut quotient $q$ is

$$q = \frac{|S|}{\min\left(|G_1|, |G_2|\right)}.$$

The minimum of $q$ over all edge separators is the **isoperimetric number**; I denote it as $i(G)$. Separator quality with respect to a graph is measured by the ratio between its cut quotient and the isoperimetric number.

With these definitions, one can define a new heuristic for extracting separators from the second eigenvector $\mathbf{u}_2$: the "best threshold cut" algorithm (see, e.g., [HK92]). Index the vertices in order based on their value from $\mathbf{u}_2$. For each index $1 \leq i \leq n-1$, consider the cut quotient of the separator produced by splitting the vertices into those with sorted index $\leq i$ and those with sorted index $> i$ (a set of vertices with the same value are considered a single vertex in this process; only cuts that separate vertices with different values need be considered). Choose the split giving the best cut quotient. I denote the best cut quotient as $q_{min}$. "Best threshold cut" is the second spectral algorithm that I analyze.

It is well known that the isoperimetric number can be bounded in terms of $\lambda_2$, the second smallest eigenvalue of the Laplacian; there is a straightforward lower bound, and an upper bound that can be shown as a consequence of Cheeger's inequality ([Alo86]). Mohar gives a slightly different upper bound on the isoperimetric number using a strong discrete version of the Cheeger inequality [Moh89]. Let $G$ be any connected graph with maximal vertex degree $\Delta$. Further, let $G$ not be any of $K_1$, $K_2$, or $K_3$, the complete graphs on 1, 2, and 3 vertices. Then

$$\frac{\lambda_2}{2} \leq i(G) \leq \sqrt{\lambda_2(2\Delta - \lambda_2)}. \tag{1.4}$$

Mohar's proof is particularly interesting because it has implications for threshold cuts: his proof implies the same upper bound holds for $q_{min}$.

Using the bounds on $i(G)$ and $q_{min}$, I show a class of graphs for which the "best threshold cut" algorithm performs poorly. In particular, I present graphs for which the isoperimetric number is at the low end of the range, while $q_{min}$ is at the upper end. For these graphs the ratio $q_{min}/i(G)$ is as large (to within a constant) as possible with respect to the bounds.

These graphs are all graph crossproducts. A crossproduct is a graph constructed from two graphs $G$ and $H$ and denoted $G \times H$. I provide a formal definition in Chapter 2; here is an informal definition: Replace every vertex in $G$ with a copy of $H$. Each edge $e$ in $G$ is then replaced by $|H|$ edges, one between each pair of corresponding vertices in the copies of $H$ that have replaced the endpoints of $e$. That is, for each edge $(v_i, v_j)$ in $G$, there is a copy $i$ and a copy $j$ of $H$. For each vertex $u$ in $H$, add an edge between vertex $u$ in copy $i$ and the vertex $u$ in copy $j$. An example is shown in Figure 1.2.

It is well known that the eigenvalues of the Laplacian of $G \times H$ are all pairwise sums of the eigenvalues of $G$ and $H$ (see e.g. [Moh88]). It is also well known that the eigenvectors of $G \times H$ can be constructed from the eigenvectors of $G$ and $H$: Let $\lambda = \mu + \nu$ be an eigenvalue of the crossproduct, where $\mu$ is an eigenvalue of $G$ with eigenvector $u$ and $\nu$ is an eigenvalue of $H$ with eigenvector $w$. Each vertex in $G \times H$ corresponds to exactly one vertex from $G$ (say $g_i$) and one vertex from $H$ (say $h_j$); thus each vertex in $G \times H$ can be written as $(g_i, h_j)$. The value of vertex $(g_i, h_j)$ in the crossproduct eigenvector is the product of the values for $g_i$ and $h_j$ from $u$ and $w$ respectively.

For an example, suppose that zero is an eigenvalue of $G$. Then in the resulting crossproduct eigenvector, all vertices in any one copy of $G$ have the same value, because all vertices in a copy of $G$ correspond to the same vertex from $H$; thus their values are the product of 1 (from the eigenvector for the 0 eigenvalue) times the $H$ eigenvector value for the $H$ vertex.

A crossproduct that causes the "best threshold cut" spectral algorithm to perform poorly is the tree-cross-path graph (Figure 1.2). A tree-cross-path graph consists of the crossproduct of a double

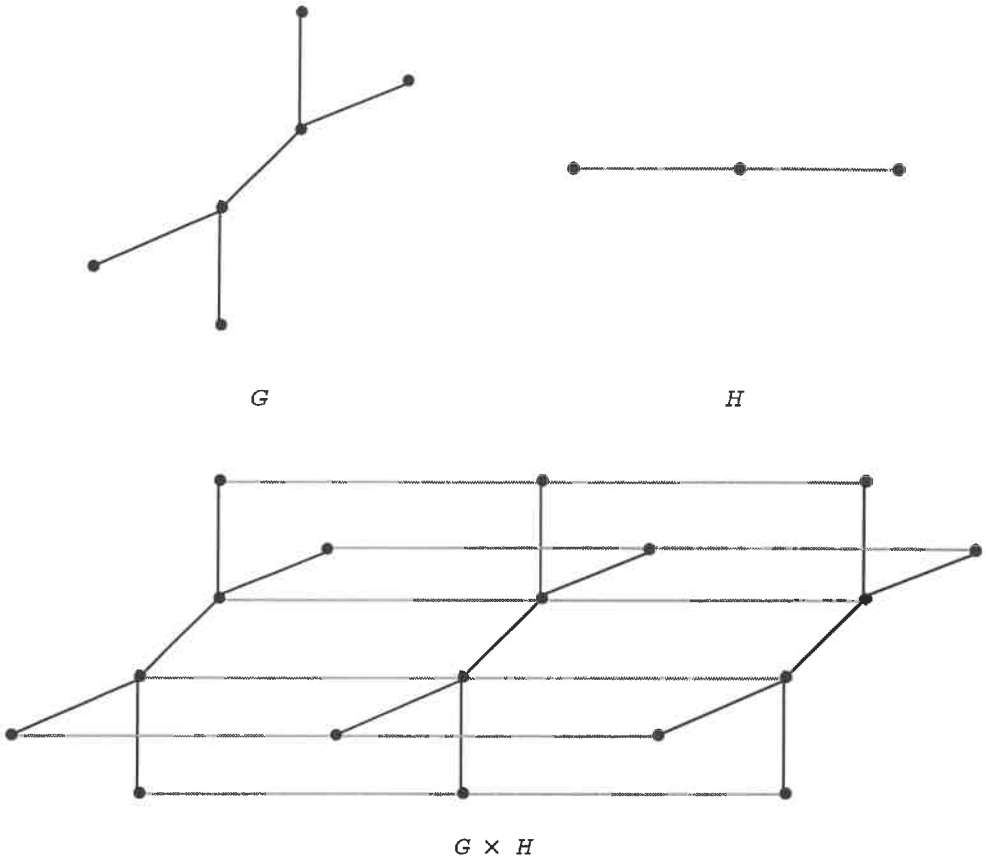



Figure 1.2: Graph Crossproduct

tree and a path graph; a double tree is a pair of complete binary trees of the same size joined by an edge between their roots. Assume that the double tree has $p$ vertices. In Chapter 2, I prove that if the path length is $c\,p^{\frac{1}{2}}$ for $c$ in the range $3.5 < c < 4$, then $\lambda_2$ of the tree-cross-path graph is the sum of the second smallest eigenvalue of the path and the 0 eigenvalue of the double tree. Thus $u_2$ of the crossproduct assigns a single value to all vertices in any copy of the double tree, and any threshold cut must separate some pair of double trees. The resulting separator has at least $p$ edges; however, cutting the edge between the roots of the complete binary trees in each double tree produces a separator of size $c\,p^{\frac{1}{2}}$. The resulting ratio of the spectral cut to the better cut is $\Theta(p^{\frac{1}{2}})$, which is as large (to within a constant factor) as possible by the bound from Mohar (given in 1.4 above).

The final algorithm I consider is based on a general definition of purely spectral separator algorithms subsuming the two preceding algorithms. An algorithm **purely spectral** if:

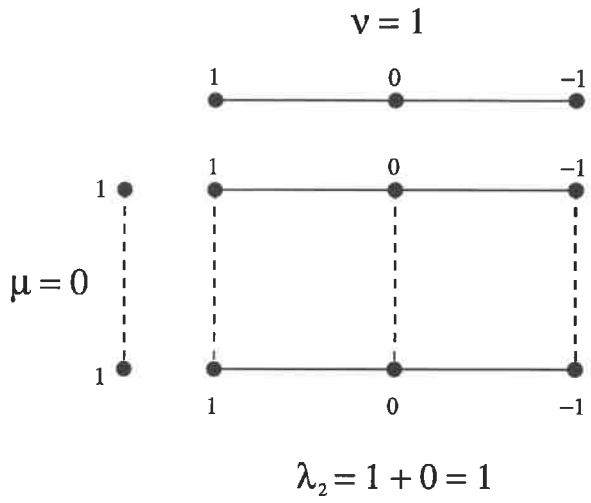



Figure 1.3: Formation of Graph Crossproduct Eigenvector

- It computes a value for each vertex using only the eigenvector components for that vertex from eigenvectors corresponding to the $k$ smallest non-zero eigenvalues (henceforth, the $k$ **smallest eigenvectors**). The function computed can be chosen arbitrarily as long as its output depends only on these inputs.
- It partitions the graph by choosing some threshold $t$ and then putting all vertices with values greater than $t$ on one side of the partition and the rest of the vertices on the other side.
- It is free to compute the break point $t$ in any way; e.g., checking the separator ratio for all possible breaks and choosing the best one is allowed.

Purely spectral algorithms suffer from similar problems to those afflicting the "best threshold cut" algorithm. In particular, if the number of eigenvectors used by the algorithm is some fixed number $k$, then the path in the tree-cross-path graph can be made long enough so that all $k$ eigenvalues correspond to eigenvalues from the path. The resulting eigenvectors all assign a single value to all vertices in a single double tree. As a result, the function applied to these values will also assign a single value to all vertices in a single double tree, and the resulting cut must again separate two copies of the double tree. The analysis is similar to that for the "best threshold cut" algorithm. Since the path length is stretched by a constant factor, the resulting ratio of cut quotient to isoperimetric number is as bad (to within a constant) as the result for the "best threshold cuts" algorithm.

If $k$ grows as a small function of $n$ (i.e., a function less that $n^{\frac{1}{4}}$), performance is still bad, though the ratio of the spectral cut to the isoperimetric number is not not as large as in the previous two cases. For sufficiently large $n$ and $0 < \epsilon < \frac{1}{4}$, there exists a bounded-degree graph $G$ on $n$ vertices such that any purely spectral algorithm using the $n^{\epsilon}$ smallest eigenvectors produces a separator $S$ for $G$ having a cut quotient greater than $i(G)$ by at least a factor of $n^{\left(\frac{1}{4}-\epsilon\right)} - 1$. The counterexample graph is still the tree-cross-path, and the analysis is similar to that used for the

"best threshold cut" algorithm.

Full details of the arguments summarized above are provided in Chapter 2.

## 1.3 The Poincaré Index Formula

The Poincaré index formula, is a topological property of embedded planar directed graphs. It is closely related to the Euler characteristic. Chapter 3 discusses the index formula itself, which has a number of implications about the types of faces and vertices that can occur in an embedded digraph; these implications in turn can be used to bound the running time of a digraph reduction algorithm. This reduction algorithm is like tree contraction in that it is a general framework on which applications can be overlaid. In particular, I present a multi-source reachability for planar DAGs built on top of the reduction algorithm that improves the running time compared to previous parallel planar digraph reachability algorithms

I will start by explaining the Poincaré index formula and its implications (this name is drawn from the Poincaré index formula from combinatorial topology; see Chapter 3 for more details).

Let $G(V, A)$ be a connected embedded planar digraph with faces $F$. A vertex of $G$ is a **source** if its indegree is zero; it is a **sink** if its outdegree is zero. The **alternation number** of a vertex is the number of direction changes of the arcs (i.e., "out" to "in" or vice versa) as we consider in cyclical order the arcs radiating from that vertex. The alternation number is always even. A source or sink has alternation number zero. A vertex is a **flow** vertex if its 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 1.4. The alternation number of a face is defined in a similar way; here alternations are

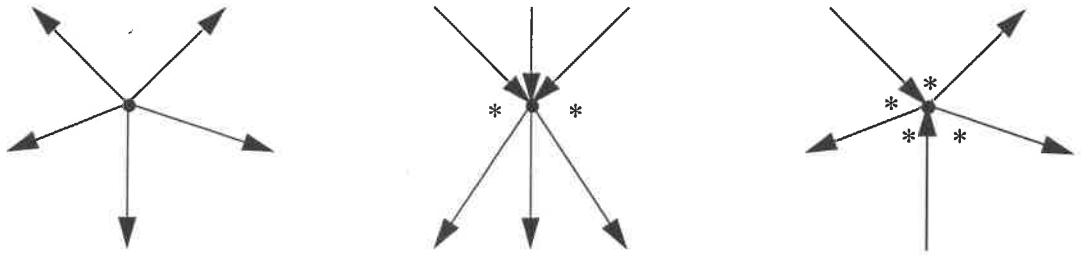



Figure 1.4: Vertex Types

defined in terms of the number of time the arcs on the boundary change direction with respect to a traversal of 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 1.5. I denote the alternation number of vertex $v$ by $\alpha(v)$, and the alternation number of face $f$ by $\alpha(f)$ (it is clear from the context whether $\alpha$ refers to a vertex or a face).

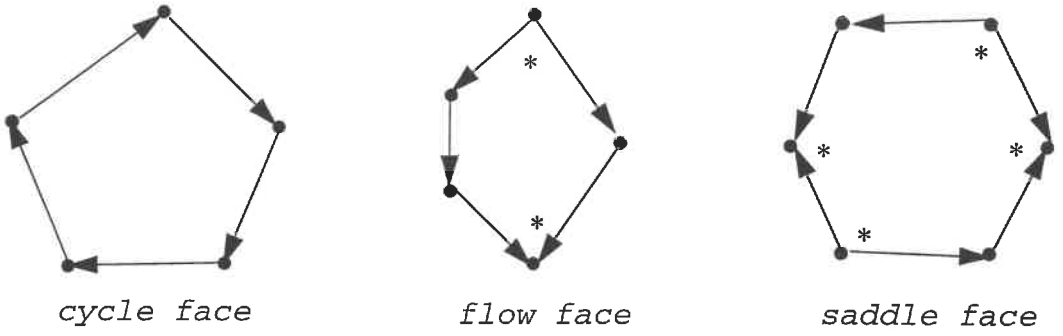



Figure 1.5: Face Types

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 I do not differentiate the notation used in these two cases.

**Theorem 1.3.1 [Poincaré Index Formula]** *For every embedded connected planar digraph, the following formula holds:*

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

This formula implies a great deal about the structure of a planar digraph embedding. For example,

- Sinks, sources, and cycle faces each contribute $-1$. These are the only structures that make negative contributions to the sums in the formula; since the total must be $-2$, it is clear that every embedded planar digraph must have at least two of them. 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; they contribute 0 to the sums in the formula. There can be an arbitrary number of such structures. It is easy to see that a flow face has two alternations on its boundary, one of which looks like a source with respect to the boundary, the other of which looks like a sink. Thus, at most one source and at most one sink can lie on the boundary of a flow face.
- Saddle vertices and saddle faces have positive indices that depend on their alternation numbers. Since the formula total 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.

### 1.3.1 The DAG Reduction Algorithm

I use the index formula to prove properties about the running time of a parallel planar DAG reduction algorithm. The algorithm consists of two parts: a general reduction algorithm that provides a method for reducing a planar DAG to a constant size and then expanding it back, and

an application overlaid on the reduction algorithm, much as specific algorithms are overlaid on the basic tree contraction mechanism. The reduction algorithm consists of a set of rules for reducing the size of planar DAG. It removes a constant fraction of the arcs in each pass through the main reduction loop. Thus, a logarithmic number of reduction steps reduce the DAG to constant size. The reduction process converts a graph into a smaller graph in order to recursively solve the desired problem. Once the problem is solved for the reduced graph, the graph is expanded back out in reverse order to generate a solution for the original graph.

The reduction rules fall into two groups: those that typically apply when the graph has relatively few sources and sinks, and those that apply when there are relatively many sources and sinks.

When the number of sources and sinks is relatively small, the graph has lots of flow faces. The main rule in this case operates on arcs that, when removed, combine two flow faces into one. This is shown in Figure 1.6. I use the Poincaré index formula to show that there are many such arcs.

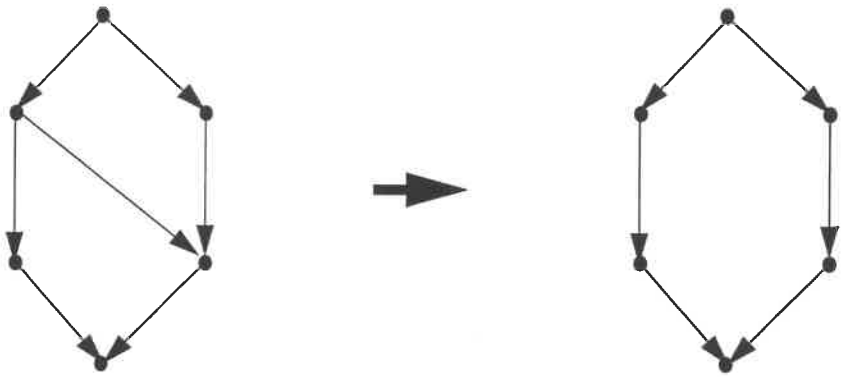


Figure 1.6: Merging Two Flow Faces

Removing arcs can affect graph connectivity, however. Therefore I associate flow faces with a data structure that maintains connectivity information consistent with the original graph. This data structure consists of a set of pointers (note: the graph represented by these pointers may be non-planar).

When there are relatively many sources and sinks, there may not be many removable arcs between flow faces. Therefore the rules operate in this case on arcs out of sources and into saddle vertices and arcs into sinks and out of saddles. "Operating" involves removing, combining, contracting, etc. The Poincaré index formula implies that if there are not many operable arcs between flow faces, then there must be many arcs to which these additional rules apply.

Here a complication arises: the connectivity pointers. Contracting arcs incident to such pointers can introduce paths that do not exist in the original graph (In particular, problems occur when a pointer enters an arc below a pointer leaving that arc, and the arc is contracted to a single vertex. Then a new path is created from these pointers when they both become incident to this vertex). I solve this problem by adding an algorithm to clean up problem pointers. For many arcs operable by the second class of rules, there are ways to determine all effects these pointers have with respect to application-specific processing; the problem pointers can then be discarded.

There is one further issue to address: applications of different rules may interfere with each other (i.e., application of one rule may make inoperable an arc previously operable by a second rule), and applications of a single rule may interfere with each other (i.e., removing two arcs from one flow face may create a saddle face). The solution is a set of conflict resolution procedures. The basic idea is, for each type of conflict, to find a maximal independent set in a graph representing the conflicts. Typically the conflict graphs have bounded degree. I show that the conflict resolution procedures do not reduce the number of arcs removed by much.

After I specify the reduction procedure, I prove that the reduction procedure given above works in $O(\log n \log^* n)$ time using $O(n)$ processors, provided that the application-specific processing takes at most constant time per reduction phase.

### 1.3.2 Planar Digraph Reachability

Once the description and analysis of the reduction process is complete, I give an application. In particular, I show the abstract reduction procedure can be extended to solve the many-source reachability problem for planar DAGs. The problem can be stated as follows: given a planar DAG and an initial set of vertices, compute the set of vertices reachable via directed paths from the initial set. I refer to the vertices reachable in this way as the solution set; the initial set is a subset of the solution set. The solution to this problem consists of a set of application-specific actions taken at various points in the reduction algorithm (These actions primarily involve vertex marking and mark propagation; initial vertices are marked, the marks are propagated over arcs and pointers, and vertices that receive a mark are listed as reachable). To show it works I introduce invariants allowing me to prove that the result is correctly computed.

The resulting algorithm tests reachability in a planar DAG in parallel on a CRCW PRAM in $O(\log n \log^* n)$ time ($O(\log n)$ time using randomization) using $O(n)$ processors. Used in conjunction with the strongly-connected components algorithm of Kao [Kao93], multiple-source reachability for general planar digraphs can be computed 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 solved in $O(\log^5 n)$ time using $O(n)$ processors.

Why is this application interesting? In particular, why is an algorithm limited to planar graphs interesting? For sequential algorithm design the two classic methods for solving problems related to reachability are breadth-first search (BFS) and depth-first search (DFS). They require time proportional to the size of the graph. Parallel polylogarithmic time algorithms for such problems currently compute the transitive closure of the graph, which requires $O(M(n))$ processors, where $M(n)$ is the number of processors needed to multiply two $n \times n$ matrices together in parallel. Since the best known value for $M(n)$ is $O(n^{2.376})$, a parallel algorithm using transitive closure for computing reachability does much more work than the corresponding sequential algorithm. For sparse graphs the situation is better, though still not optimal: Ullman and Yannakakis give a probabilistic parallel algorithm running in $O(\sqrt{n})$ time using $n$ processors [UY90]. This blow-up in the amount of work for parallel algorithms, often referred to as the transitive closure bottleneck, 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 [Kao93], Kao and Shannon [KS89] [KS93],

and Kao and Klein [KK90] showed that the reachability problem and many related problems could be solved in polylogarithmic time using only a linear number of processors for planar digraphs. The planar reachability problem for multiple start vertices is specifically addressed in [KK90]. The methods in that paper involve a series of reductions between related problems; each reduction introduces more logarithmic factors to the running time. In the end it takes $O(\log^5 n)$ time to solve this problem.

Directed planar graphs are important for at least two reasons. First, this class includes several important subclasses including tree and series parallel graphs. Second, the flow graphs for many structured programming languages without function calls are planar. One goal for future work is the development of basic algorithms for a class of planar graphs so that a theory of planar flow graphs could be based on it.

The specifics of the general reduction procedure and its application in multi-source DAG reachability are provided in Chapter 3.

## 1.4 Future Work

The thesis concludes with Chapter 4, which covers extensions and potential future work.

Future work for spectral separators includes research on the following questions:

- How can spectral partitioning algorithms be modified to improve their performance?
- How can the eigenvector properties of specific graphs be used to generate a new taxonomy of graphs that relates graph properties to the quality of spectral separators?
- How can the techniques from Chapter 2 be used in other ways?

The work on the Poincaré index formula and on planar digraph algorithms offers the following areas for research:

- Extending the reduction algorithm from planar DAGs to all planar digraphs.
- Using the reduction algorithm to implement other planar digraph algorithms.
- Using algorithms built on the reduction algorithm in practical applications. This may require modifying the algorithm for easier implementation and faster execution.

## 1.5 Contributions

The contributions of this thesis are:

- New properties about the structure of Laplacian eigenvectors.
- Analysis of three commonly-used spectral separator algorithms showing that there are graphs similar to those that arise in practice for which spectral separator algorithms produce poor separators.

- The Poincaré index formula.
- A general parallel planar DAG reduction algorithm that can be used in the design of algorithms.
- A parallel multi-source reachability algorithm for planar digraphs with better asymptotic performance than previous algorithms.

# Chapter 2

# On the Quality of Spectral Separators

## 2.1 Introduction

Spectral methods (i.e., methods that use the eigenvalues and eigenvectors of a matrix representation of a graph) are widely used to compute graph separators. Typically, the Laplacian matrix is used; the Laplacian $B$ of a graph $G$ on $n$ vertices is the $n \times n$ matrix with the degrees of the vertices of $G$ on the diagonal, and entry $b_{ij} = -1$ if $G$ has the edge $(v_i, v_j)$ and 0 otherwise. The eigenvector $\mathbf{u}_2$ corresponding to $\lambda_2$ (the second-smallest eigenvalue of $B$) is computed, and the vertices of the graph are partitioned according to the values of their corresponding entries in $\mathbf{u}_2$ [PSL90, HK92]. The goal is to compute a small separator; that is, as few edges or vertices as possible should be deleted from the graph to achieve the partition. Additionally, the sizes of the resulting components should be roughly comparable.

Although spectral methods are popular, there is little analysis of how well they do in producing small separators. Instead, it is usually claimed that such methods "work well in practice," and tables of results for specific examples are often included in papers (see e.g. [PSL90]). Thus there is little guidance for someone wishing to compute separators as to whether or not this technique is appropriate. Ideally, practitioners should have a characterization of classes of graphs for which spectral separator techniques work well; this characterization might be in terms of how far the computed separators can be from optimal. This chapter represents a first step in this direction. I consider two spectral separation algorithms that partition the vertices on the basis of the values of their corresponding entries in $\mathbf{u}_2$, and provide counterexamples for which each of the algorithms produces poor separators. I further consider a generalized definition of spectral methods that allows the use of more than one of the eigenvectors corresponding to the smallest non-zero eigenvalues, and show that there are graphs for which any such algorithm does poorly.

The first algorithm is a popular technique that consists of bisecting a graph by partitioning the vertices into two equal-sized sets based on each vertex's entry in the eigenvector $\mathbf{u}_2$. A graph in this first counterexample class looks like a ladder with the top $2/3$ of its rungs kicked out (see Figure 2.1); a straightforward spectral bisection algorithm cuts the remaining rungs, whereas the optimal bisection is made by cutting across the ladder above the remaining rungs. (I refer to this graph as the "roach graph" because its outline looks roughly like the body of a cockroach and two

long antennae.)

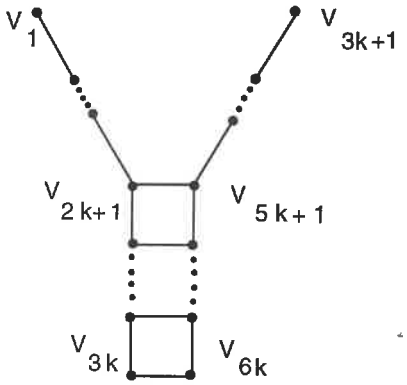



Figure 2.1: The Roach Graph

The spectral bisection algorithm can be modified to generate a better separator for the roach graph. Some modifications are presented in [HK92]; they still use a partition based on $\mathbf{u}_2$. I consider a simple spectral separator algorithm, the "best threshold cut" algorithm, based on the most general of these suggested modifications, and present a second class of graphs that defeats this algorithm. This class consists of crossproducts of path graphs and graphs consisting of a pair of complete binary trees connected by an edge between their roots.

Finally, I consider a more general definition of purely spectral separator algorithms that subsumes the two preceding algorithms. This definition allows the use of some specified number of eigenvectors corresponding to the smallest eigenvalues of the Laplacian. For any such algorithm that uses a fixed number of eigenvectors I show there are graphs for which it does no better than using the "best threshold cut" algorithm. Further, the separator produced when the "best threshold cut" algorithm is applied to these graphs is as bad as possible (to within a constant) with respect to bounds on the size of the separators produced. I also show that if a purely spectral algorithm uses up to $n^\epsilon$ eigenvectors for $0 < \epsilon < \frac{1}{4}$, there exist graphs for which the algorithm fails to find a separator with a **cut quotient** (i.e., the ratio between the number of edges cut and the size of the smaller set in the vertex partition) within $n^{\frac{1}{4}-\epsilon} - 1$ of the optimum (the optimum cut quotient is called the **isoperimetric number**). I also argue that the counterexample graphs can be extended to graphs that could conceivably be used as three-dimensional finite-element meshes – that is, graphs that could be encountered in practice.

This thesis makes one additional contribution: While the counterexamples have simple structures and intuitively might be expected to cause problems for spectral separator algorithms, the challenge is to provide good bounds on $\lambda_2$ for these graphs. For this purpose I have developed theorems about the spectra of graphs with particular symmetries that exist in the counterexamples.

Specifics are given in the text that follows. Section 2.2 gives the history of spectral methods and the details of the algorithms discussed in this paper. Section 2.3 gives some graph and matrix terminology, and presents some useful facts about Laplacians. Section 2.4 gives the counterexample for the spectral bisection algorithm; Section 2.5 gives the counterexample for the "best threshold cut" algorithm. Section 2.6 discusses the generalized definition of spectral separator algorithms, and

shows that there are graphs for which any such algorithm performs poorly. Section 2.7 concludes the discussion of spectral methods; it summarizes the contributions of this work. Future work is discussed in Chapter 4.

## 2.2 Spectral Methods for Computing Separators

The roots of spectral partitioning go back to Hoffmann and Donath [DH73], who proved a lower bound on the size of the minimum bisection of a graph, and Fiedler [Fie73][Fie75], who explored the properties of $\lambda_2$ and its associated eigenvector for the Laplacian. There has been much subsequent work, including Barnes's partitioning algorithm [Bar82], Boppana's work that included a stronger lower bound on the minimum bisection size [Bop87], and the particular bisection and graph partitioning problems considered in this paper [HK92] [PSL90] [Sim91]. (Note that spectral methods have not been limited to graph partitioning; work has been done using the spectrum of the adjacency matrix in graph coloring [AG84] and using the Laplacian spectrum to prove theorems about expander graph and superconcentrator properties [AM85] [Alo86] [AGM87]. The work on expanders has explored the relationship of $\lambda_2$ to the isoperimetric number; Mohar has given an upper bound on the isoperimetric number using a strong discrete version of the Cheeger inequality [Moh89]. Reference [CDS79] is a book-length treatment of graph spectra, and it predates many of the results cited above.)

A basic way of computing a graph bisection using spectral information is presented in [PSL90]. I refer to this algorithm as **spectral bisection**. Spectral bisection works as follows:

- Represent $G$ by its Laplacian $B$, and compute $\mathbf{u}_2$, the eigenvector corresponding to $\lambda_2$ of $B$.
- Assign each vertex the value of its corresponding entry in $\mathbf{u}_2$. This is the **characteristic valuation** of $G$.
- Compute the median of the elements of $\mathbf{u}_2$. Partition the vertices of $G$ as follows: the vertices whose values are less than or equal to the median element form one part of the partition; the rest of the vertices form the other part. The set of edges between the two parts forms an edge separator.
- If a vertex separator is desired, it is computed from the edge separator as described in the next section.

Since the graph bisection problem is NP-complete [GJ79], spectral bisection may not give an optimum result. That is, spectral bisection is a heuristic method. A number of modifications have been proposed that may improve its performance. These modified heuristics may give splits other than bisections. In such cases, one can use the cut quotient or separator ratio (the **separator ratio** is the ratio between the number of edges cut and the product of the sizes of the two sets in the vertex partition) to judge how close the split is to optimal. Computing the separator with the minimum ratio is NP-hard (see, e.g., [LR88]). The following modifications, all of which use the characteristic valuation, are presented in [HK92]:

- Partition the vertices based on the signs of their values;

- Look for a large gap in the sorted list of eigenvector components, and partition the vertices according to whether their values are above or below the gap; and
- Sort the vertices according to value. For each index $1 \leq i \leq n-1$, consider the ratio for the separator produced by splitting the vertices into those with sorted index $\leq i$ and those with sorted index $> i$. Choose the split that provides the best separator ratio.

Note that the last idea subsumes the first two. I consider a variant of this algorithm below. Since this algorithm does not specify what to do when multiple vertices have the same value, I restrict it to consider only splits between vertices with different values (such cuts are called **threshold cuts**). This restricted version is the **"best threshold cut" algorithm**; the slight change from the definition above does not alter its performance with respect to the counterexamples below (other than slightly simplifying the analysis).

Also note that the idea of cutting at an arbitrary point along the sorted order can be extended to choosing two split points, where the corresponding partitions are the vertices with values between the split points, and those with values above the upper or below the lower split point. Again, the pair yielding the best ratio is chosen.

The algorithms mentioned so far have only used the eigenvector $\mathbf{u}_2$. Another possibility is to look at partitions generated by the set of eigenvectors for some number of smallest eigenvalues: for each vertex, a value is assigned by computing a function of that vertex's eigenvector components. Partitions are then generated in the same way as they are for $\mathbf{u}_2$ in the various algorithms given above.

Given the variety of heuristics cited above, it would be nice to know which ones work well for which classes of graphs. It would be particularly useful if it were possible to state reasonable bounds on the performance of these heuristics for classes of graphs commonly used in practice (e.g., planar graphs, planar graphs of bounded degree, three-dimensional finite element meshes, etc.). Unfortunately, this is not the case. I start by proving that spectral bisection may produce a bad separator for a bounded-degree planar graph in Section 2.4; first, however, I need to introduce some terminology and background results.

## 2.3 Terminology, Notation, and Background Results

I assume that the reader is familiar with the basic definitions of graph theory (in particular, for undirected graphs), and with the basic definitions and results of matrix theory. A graph consists of a set of vertices $V$ and a set of edges $E$; I denote the vertices (respectively edges) of a particular graph $G$ as $V(G)$ (respectively $E(G)$) if there is any ambiguity about which graph is referred to. The notation $|G|$ is sometimes be used as a shorthand for $|V(G)|$. When it is clear which graph is referred to, I use $n$ to denote $|V|$.

Capital letters represent matrices and bold lower-case letters represent vectors. For a matrix $A$, $a_{ij}$ or $[A]_{ij}$ represents the element in row $i$ and column $j$; for the vector $\mathbf{x}$, $x_i$ or $[\mathbf{x}]_i$ represents the $i^{\text{th}}$ entry in the vector. The notation with square brackets is useful in cases where adding subscripts to lower-case letters is awkward (e.g., where the matrix or vector name is already subscripted). The notation $\mathbf{x} = 0$ indicates that all entries of the vector $\mathbf{x}$ are zero; $\vec{1}$ indicates the vector that has 1 for

every entry. For ease of reference, the eigenvalues of an $n \times n$ matrix are indexed in non-decreasing order. $\lambda_1$ represents the smallest eigenvalue, and $\lambda_n$ the largest. For $1 < i < n$, $\lambda_{i-1} \leq \lambda_i \leq \lambda_{i+1}$. The notation $\lambda_i(A)$ (respectively $\lambda_i(G)$) indicates the $i^{\text{th}}$ eigenvalue of matrix $A$ (respectively of the Laplacian of graph $G$) if there is any ambiguity about which matrix (respectively graph) the eigenvalue belongs to. $\mathbf{u}_i$ represents the eigenvector corresponding to $\lambda_i$.

The term **path graph** denotes a tree that has exactly two vertices of degree one. That is, a path graph is a graph consisting of exactly its maximal path.

The **crossproduct** of two graphs $G$ and $H$ (denoted $G \times H$) is a graph on the vertex set $\{(u, v) \mid u \in V(G), v \in V(H)\}$, with $((u, v), (u', v'))$ in the edge set if and only if either $u = u'$ and $(v, v') \in E(H)$ or $v = v'$ and $(u, u') \in E(G)$. It is easy to see that $G \times H$ and $H \times G$ are isomorphic. One way to think of a graph crossproduct is as follows: Replace every vertex in $G$ with a copy of $H$. Each edge $e$ in $G$ is then replaced by $|H|$ edges, one between each pair of corresponding vertices in the copies of $H$ that have replaced the endpoints of $e$. That is, for each edge $(v_i, v_j)$ in $G$, there is a copy $i$ and a copy $j$ of $H$. For each vertex $u$ in $H$, add an edge between vertex $u$ in copy $i$ and the vertex $u$ in copy $j$. An example is shown in the figure below.

For a connected graph $G$, an **edge separator** is a set $S$ of edges that, if removed, breaks the graph into two (not necessarily connected) components $G_1$ and $G_2$ such that there are no edges between $G_1$ and $G_2$. (An edge separator is defined to be a minimal set with respect to the particular $G_1$ and $G_2$.) A **vertex separator** is a set $S$ of vertices such that if these vertices and all incident edges are removed, the graph is broken into two components $G_1$ and $G_2$ such that there are no edges between $G_1$ and $G_2$ (again, such a separator is a minimal such set). The goal in finding separators is to find a small separator that breaks the graph into two fairly large pieces; often this notion of "large pieces" is expressed as a restriction that the number of vertices in either component be at least some specified fraction of the number of vertices in $G$. For edge separators, this is stated more generally in terms of the **separator ratio** $\rho$, defined as $|S|/\left(|G_1| \cdot |G_2|\right)$. The **optimum ratio separator** $\rho_{opt}$ is the one that minimizes the separator ratio over all separators for a particular graph [LR88].

A related concept (again, for edge separators) is the **isoperimetric number** $i(G)$, defined as:

$$\min_S \left( \frac{|S|}{\min\left(|G_1|, |G_2|\right)} \right).$$

I refer to the quantity $|S|/\min\left(|G_1|, |G_2|\right)$ as the **cut quotient** for the edge separator $S$. It is easy to see that $n\rho_{opt} > i(G) \geq n\rho_{opt}/2$. As noted in the chapter introduction, finding a cut with the minimum separator ratio or with a cut quotient equal to the isoperimetric number is NP-hard.

It is well known that an edge separator $S$ can easily be converted into a vertex separator $S'$ by considering the bipartite graph induced by $S$ (where the parts of the bipartition are determined by the components $G_1$ and $G_2$), and setting $S'$ to be a minimum edge cover for that graph.

Graphs can be represented by matrices. For example, the **adjacency matrix** $A$ of a graph $G$ is defined as $a_{ij} = 1$ if and only if $(v_i, v_j) \in E(G)$; $a_{ij} = 0$ otherwise. (For such representations I assume that the vertices are numbered to correspond to the indices used in the matrix.)

A common matrix representation of graphs is the **Laplacian**. Let $D$ be the matrix with $d_{ii} = \text{degree}(v_i)$ for $v_i \in V(G)$, and all off-diagonal entries equal to zero. Let $A$ be the adjacency matrix for $G$ as defined above. Then the Laplacian of $G$ is the matrix $B = D - A$.

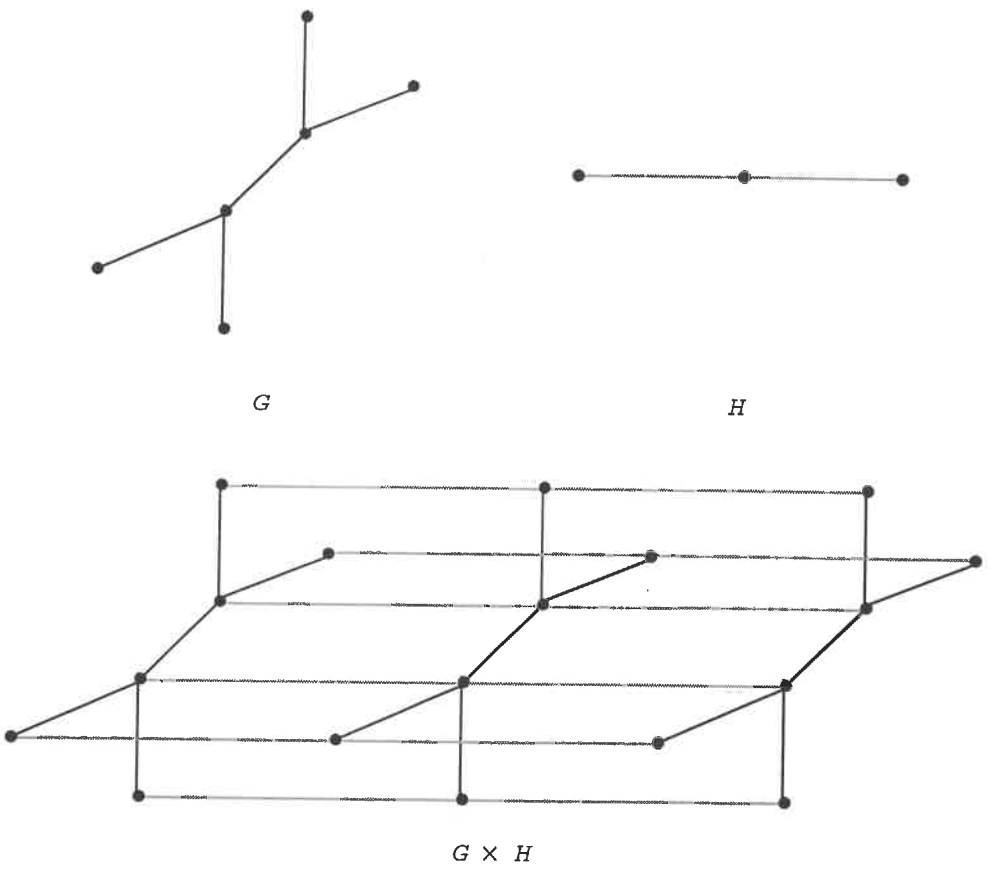



Figure 2.2: A Graph Crossproduct Example

The following are some useful facts about the Laplacian matrix:

- The Laplacian is symmetric positive semidefinite, so all its eigenvalues are greater than or equal to 0 (see e.g. [Moh88]).
- A graph $G$ is connected if and only if 0 is a simple eigenvalue of the Laplacian of $G$ (see e.g. [Moh88]).
- The following characterization of $\lambda_2$ holds (see e.g. [Fie73]):

$$\lambda_2 = \min_{\mathbf{x} \perp \vec{1}} \frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}}.$$

- If $G$ is a crossproduct of two graphs $G$ and $H$, then the eigenvalues of the Laplacian of $G$ are all pairwise sums of the eigenvalues of $G$ and $H$ (see e.g. [Moh88]).

- For any vector $\mathbf{x}$ and Laplacian $B$ of graph $G$, the following holds (see e.g. [HK92]):

$$\mathbf{x}^T B\mathbf{x} = \sum_{(v_i,v_j)\in E(G)} (x_i - x_j)^2 \tag{2.1}$$

- For a graph $G$ and its Laplacian $B$, $\lambda_2$ of $B$ figures in upper and lower bounds on the isoperimetric number for $G$ [Moh89]. In particular, if $G$ is not one of $K_1$, $K_2$, or $K_3$ (the complete graphs on 1, 2, and 3 vertices respectively), then:

$$\frac{\lambda_2}{2} \leq i(G) \leq \sqrt{\lambda_2(2\Delta - \lambda_2)}, \tag{2.2}$$

  where $\Delta$ is the maximum degree of any vertex in $G$. This implies the size of any bisection is at least $\frac{n\lambda_2}{4}$.

The proof of the upper bound in (2.2) has interesting implications about the threshold cuts based on the second eigenvector. For any connected graph $G$, consider the characteristic valuation. The vertices of $G$ receive $k \leq n$ distinct values; let $t_1 > t_2 > \ldots > t_k$ be these values. For each threshold $t_i$, $i < k$, divide the vertices into those with values greater than $t_i$, and those with values less than or equal to $t_i$. Compute the cut quotient $q_i$ for each such cut, and let $q_{min}$ be the minimum over all $q_i$'s. The following theorem can be derived from the proof of Theorem 4.2 in [Moh89]:

**Theorem 2.3.1** *Let $G$ be a connected graph with maximal vertex degree $\Delta$ and second smallest eigenvalue $\lambda_2$. If $G$ is not any of $K_1$, $K_2$, or $K_3$, then*

$$\frac{\lambda_2}{2} \leq q_{min} \leq \sqrt{\lambda_2(2\Delta - \lambda_2)}.$$

This can be extended to the separator ratio for the best $u_2$ cut in the obvious way.

A **weighted** graph is a graph for which a real value $w_i$ is associated with each vertex $v_i$, and a real, nonzero weight $w_{ij}$ is associated with each edge $(v_i, v_j)$ (a zero edge weight indicates the lack of an edge). For my analysis, I need a matrix representation for weighted graphs. Fiedler extended the notion of the Laplacian to graphs with positive edge weights [Fie75]; he referred to this representation as the **generalized Laplacian**. However, I need an even more general representation that can be used for any weighted graph. Hence I define the **standard matrix representation** $B$ of a weighted graph $G$ as follows: $B$ has $b_{ii} = w_i$; for $i \neq j$ and $(v_i, v_j) \in E(G)$, $b_{ij} = -w_{ij}$, and $b_{ij} = 0$ otherwise. Note that the Laplacian matrix of a graph is also the standard matrix representation of the graph with vertex weights set to be the degrees of the vertices, and all edge weights set to 1.

Note that the standard matrix representation of any weighted graph is a real symmetric matrix, and that any such matrix can be represented as a specific weighted graph. Any theorems about symmetric, real matrices apply to standard matrix representations of graphs. The following two theorems about the interlacing of eigenvalues are particularly useful when applied to standard matrix representations.

The **First Interlacing Property** states that if $A_r$ denotes the leading $r \times r$ principal submatrix of an $n \times n$ symmetric matrix $A$, then for $r = 1 : n-1$ the following holds:

$$\lambda_{r+1}(A_{r+1}) \geq \lambda_r(A_r) \geq \lambda_r(A_{r+1}) \geq \ldots$$
$$\ldots \geq \lambda_2(A_{r+1}) \geq \lambda_1(A_r) \geq \lambda_1(A_{r+1})$$

(the preceding statement is from page 411 of reference [GL89]; the proof is in [Wil65]).

The **Second Interlacing Property** is stated as follows: Suppose $B = A + \alpha \mathbf{c}\mathbf{c}^T$, where $A$ is a real symmetric $n \times n$ matrix, $\mathbf{c}$ is a real unit-length vector, and $\alpha$ is real and greater than 0. Then for all $i$, $1 \leq i \leq n-1$

$$\lambda_i(A) \leq \lambda_i(B) \leq \lambda_{i+1}(A)$$

(the preceding statement is a restricted version of a theorem from page 412 of reference [GL89]; the proof is in [Wil65]). The Second Interlacing Property implies that if an edge $e$ is added to a graph $G$ to produce the graph $G'$, then $\lambda_2(G) \leq \lambda_2(G')$.

### 2.3.1 The Structure of Laplacian and Standard Matrix Representation Eigenvectors

The theorems and lemmas presented in this section are useful in proving results about the eigenvectors of the families of graphs presented in later sections. The proofs of some of these results are long and technical; a reader who is interested only in understanding the counterexamples and their implications presented later in this report is advised to look at the theorem statements and examples, and to skip the proofs (although the rules for constructing odd and even components at the end of Theorem 2.3.4 may be of interest to some readers).

The first set of results concerns eigenvalues of Laplacians of graphs with automorphisms of order 2. A **graph automorphism** is a permutation $\phi$ on the vertices of the graph $G$ such that $(v_i, v_j) \in E(G)$ if and only if $(v_{\phi(i)}, v_{\phi(j)}) \in E(G)$. The **order** of a graph automorphism is the order of the permutation on the vertices.

For weighted graphs, there are two additional conditions: the weights of vertices $v_i$ and $v_{\phi(i)}$ must be equal for all $i$, and the weights of edges $(v_i, v_j)$ and $(v_{\phi(i)}, v_{\phi(j)})$ must be equal.

The next two theorems concern the structure of eigenvectors with respect to automorphisms of order 2. They hold both for Laplacians of graphs under the standard definition of automorphism, and for standard matrix representations of weighted graphs under the definition of automorphisms for weighted graphs.

Let $G$ be a graph with an automorphism $\phi$ of order 2 and Laplacian $B$. A vector $\mathbf{x}$ that has $x_i = x_{\phi(i)}$ for all $i$ in the range $1 \leq i \leq n$ is an **even** vector with respect to the automorphism $\phi$; an **odd** vector $\mathbf{y}$ has $y_i = -y_{\phi(i)}$ for all $i$. It is easy to show that for any even vector $\mathbf{x}$ and odd vector $\mathbf{y}$ (both with respect to $\phi$), $\mathbf{x}$ and $\mathbf{y}$ are orthogonal.

**Theorem 2.3.2 [Even-Odd Eigenvector Theorem]** *Let $B$ be the Laplacian of a graph $G$ that has an automorphism $\phi$ of order 2. Then there exists a complete set $\mathcal{U}$ of orthogonal eigenvectors of $B$ such that any eigenvector $\mathbf{u} \in \mathcal{U}$ is either even or odd with respect to $\phi$. This also holds if $G$ is a weighted graph, $B$ the standard matrix representation of $G$, and $\phi$ a weighted graph automorphism of order 2.*

**Proof:** Let $P$ be the permutation matrix that corresponds to the automorphism $\phi$. Then $P^T B P = B$. If $\mathbf{u}$ is an eigenvector of $B$ with eigenvalue $\lambda$, then so is $P\mathbf{u}$. The following holds by the definition of automorphism:

$$\left(P^T B P\right) \mathbf{u} = B\mathbf{u} = \lambda \mathbf{u}.$$

Since the automorphism is of order 2, $PP = I$ and $P^T = P^{-1} = P$. Multiplying each side of the

equation above by $P$ thus gives

$$B(P\mathbf{u}) = P(\lambda\mathbf{u}) = \lambda(P\mathbf{u}).$$

For an even vector $\mathbf{x}$, $P\mathbf{x} = \mathbf{x}$; for an odd vector $\mathbf{y}$, $P\mathbf{y} = -\mathbf{y}$.

$P$ allows us to uniquely decompose any vector $\mathbf{x}$ into an odd component $\mathbf{x}_{odd}$ and an even component $\mathbf{x}_{even}$ as follows:

$$\mathbf{x}_{odd} = \frac{\mathbf{x} - P\mathbf{x}}{2}, \text{ and } \mathbf{x}_{even} = \frac{\mathbf{x} + P\mathbf{x}}{2}.$$

For any non-zero $\mathbf{x}$, at least one of the even or odd parts must also be non-zero.

Let $\mathcal{U}'$ be any complete set of eigenvectors of $B$. For an eigenvector $\mathbf{u} \in \mathcal{U}'$, it is easy to see that a non-zero even or odd component is an eigenvector for the same eigenvalue. Since $\mathbf{u}_{odd} + \mathbf{u}_{even} = \mathbf{u}$, the set of odd and even eigenvectors resulting from decomposing all the eigenvectors spans the same space as $\mathcal{U}$. This implies the claimed result.

The proof generalizes to weighted graphs.

□

**Corollary 2.3.3** *Let $B$ be the standard matrix representation of a weighted graph $G$ that has one or more automorphisms of order 2. Then the eigenvector for any simple eigenvalue is either even or odd with respect to every such automorphism.*

**Proof:** Let $\mathbf{u}$ be the eigenvector for some simple eigenvalue $\lambda$. Consider the decomposition of $\mathbf{u}$ into odd and even parts with respect to some automorphism $\phi$ with order 2. If both parts were non-zero, they would be orthogonal and eigenvectors for $\lambda$. Therefore either the odd part or the even part must be zero.

□

Since Laplacians can be considered to be standard matrix representations given the right weight assignments, the preceding result also holds for Laplacians.

**Theorem 2.3.4 [Even-Odd Graph Decomposition Theorem]** *For every weighted graph $G$ with standard matrix representation $B$ and an automorphism $\phi$ of order 2, there exist two smaller weighted graphs $G_{odd}$ and $G_{even}$ (the odd and even components respectively) such that the eigenvalues of $B_{odd}$ (the standard matrix representation of $G_{odd}$) are odd eigenvalues of $B$, the eigenvalues of $B_{even}$ (the standard matrix representation of $G_{even}$) are even eigenvalues of $B$, and $|V(G_{odd})| + |V(G_{even})| = |V(G)|$. Further, a complete set of eigenvectors of $B$ can be constructed from the eigenvectors of $B_{odd}$ and $B_{even}$ in a straightforward way.*

**Proof:** I demonstrate a similarity transform that when applied to $B$ gives a reducible matrix with two blocks corresponding to $B_{odd}$ and $B_{even}$. First I introduce some notation.

The vertices of $G$ can be divided into two disjoint sets on the basis of how $\phi$ operates on them. Let $V_f$ be the set of vertices $v_i$ such that $\phi(i) = i$ (i.e., the vertices fixed by $\phi$), and let $V_m$ be the set of vertices $v_j$ such that $\phi(j) \neq j$ (i.e., the vertices moved by $\phi$). $V_m$ consists of vertices in orbits of length 2. I call a subset of $V_m$ that consists of exactly one vertex from each such orbit a **representative set** and denote it $V_r$. In the rest of the proof assume that a particular $V_r$ has been arbitrarily specified. I use $n_f$, $n_m$, and $n_r$ respectively to denote the number of vertices in each of these sets.

Without loss of generality, number the vertices in the following way: the vertices in $V_f$ are numbered 1 through $n_f$; the vertices in $V_r$ are numbered from $n_f + 1$ to $n_f + n_r$. If $v_i \in V_r$, set $\phi(i) = i + n_r$; that is, the vertices in $V_m \setminus V_r$ are numbered $n_f + n_r + 1$ to $n$ in the same order as the vertices in $V_r$ with which they share orbits. Using this ordering and the definition of the automorphism, $B$ can be written in the following block form

$$B = \begin{bmatrix} F & E_{fr} & E_{fr\bullet} \\ E_{fr}^T & R & E_{r\phi(r)} \\ E_{fr}^T & E_{r\phi(r)} & R \end{bmatrix},$$

where

- $F$ is an $n_f \times n_f$ submatrix containing the diagonal entries for the vertices in $V_f$ and the entries for edges between pairs of vertices in $V_f$;
- $R$ is an $n_r \times n_r$ submatrix containing the diagonal entries for the vertices in $V_r$ and the entries for edges between pairs of vertices in $V_r$ (recall that the definition of an automorphism and the vertex numbering together imply that the same submatrix applies for the vertices in $V_m \setminus V_r$);
- $E_{fr}$ contains the entries of $B$ for edges between vertices in $V_f$ and $V_r$ (the vertex numbering plus the automorphism condition that $(v_i, v_j)$ is an edge of $G$ if and only if $(v_{\phi(i)}, v_{\phi(j)})$ is also an edge imply that the submatrix with entries for edges between $V_f$ and $V_m \setminus V_r$ are the same); and
- $E_{r\phi(r)}$ contains the entries of $B$ for edges between vertices in $V_r$ and $V_f \setminus V_r$ (again, the vertex numbering, the condition that edges are preserved under automorphism, and the symmetry of $B$ imply that $E_{r\phi(r)} = E_{r\phi(r)}^T = E_{\phi(r)r}$).

I now define the orthogonal matrix $T$ used to transform $B$. $T$ has the following form:

$$T = \begin{bmatrix} I_{n_f} & 0 & 0 \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{1}{\sqrt{(2)}} I_{n_r} \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{-1}{\sqrt{(2)}} I_{n_r} \end{bmatrix},$$

where the $I$'s are identity matrices with the dimension specified in the subscript. $B$ is transformed as follows:

$$\begin{aligned} B' &= T^T B T \\ &= \begin{bmatrix} I_{n_f} & 0 & 0 \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{1}{\sqrt{(2)}} I_{n_r} \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{-1}{\sqrt{(2)}} I_{n_r} \end{bmatrix} \begin{bmatrix} F & E_{fr} & E_{fr} \\ E_{fr}^T & R & E_{r\phi(r)} \\ E_{fr}^T & E_{r\phi(r)} & R \end{bmatrix} \begin{bmatrix} I_{n_f} & 0 & 0 \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{1}{\sqrt{(2)}} I_{n_r} \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{-1}{\sqrt{(2)}} I_{n_r} \end{bmatrix} \\ &= \begin{bmatrix} I_{n_f} & 0 & 0 \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{1}{\sqrt{(2)}} I_{n_r} \\ 0 & \frac{1}{\sqrt{(2)}} I_{n_r} & \frac{-1}{\sqrt{(2)}} I_{n_r} \end{bmatrix} \begin{bmatrix} F & \sqrt{2} E_{fr} & 0 \\ E_{fr}^T & \frac{1}{\sqrt{(2)}}(R + E_{r\phi(r)}) & \frac{1}{\sqrt{(2)}}(R - E_{r\phi(r)}) \\ E_{fr}^T & \frac{1}{\sqrt{(2)}}(R + E_{r\phi(r)}) & \frac{1}{\sqrt{(2)}}(-R + E_{r\phi(r)}) \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} F & \sqrt{2}E_{fr} & 0 \\ \sqrt{2}E_{fr}^T & R + E_{r\phi(r)} & 0 \\ 0 & 0 & R - E_{r\phi(r)} \end{bmatrix}.$$

Note that the resulting matrix is reducible. That is, when viewed as a weighted graph, that graph has two components. I show that the blocks of this matrix correspond to $B_{even}$ and $B_{odd}$ as follows:

$$B_{even} = \begin{bmatrix} F & \sqrt{2}E_{fr} \\ \sqrt{2}E_{fr}^T & R \end{bmatrix} \text{ and } B_{odd} = R - E_{r\phi(r)}.$$

Since $B'$ is similar to $B$, it has a number of useful properties. Because $B'$ is reducible, every eigenvalue of $B_{odd}$ is an eigenvalue $B'$; likewise every eigenvalue of $B_{even}$ is an eigenvalue $B'$. By similarity, they are also eigenvalues of $B$.

Now consider an eigenvector $\mathbf{u}$ of $B_{even}$. Define $\mathbf{v}$ as follows: for $1 \leq i \leq n_f + n_r$ let $v_i = u_i$; let $v_i = 0$ otherwise. $\mathbf{v}$ is obviously an eigenvector of $B'$. Multiplication by the matrix $T$ transforms $\mathbf{v}$ into an eigenvector $\mathbf{w}$ of $B$:

$$\mathbf{w} = T\mathbf{v} = \begin{bmatrix} I_{n_f} & 0 & 0 \\ 0 & \frac{1}{\sqrt{(2)}}I_{n_r} & \frac{1}{\sqrt{(2)}}I_{n_r} \\ 0 & \frac{1}{\sqrt{(2)}}I_{n_r} & \frac{-1}{\sqrt{(2)}}I_{n_r} \end{bmatrix} \begin{bmatrix} \mathbf{v}_f \\ \mathbf{v}_r \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{v}_f \\ \frac{1}{\sqrt{(2)}}\mathbf{v}_r \\ \frac{1}{\sqrt{(2)}}\mathbf{v}_r \end{bmatrix}$$

By the vertex numbering, it is easy to see this is an even vector. Since $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ all have the same eigenvalue $\lambda$, the claim about eigenvalues of $B_{even}$ corresponding to even eigenvalues of $B$ holds. It is easy to show that if two eigenvectors $\mathbf{u_1}$ and $\mathbf{u_2}$ of $B_{even}$ are orthogonal, then the corresponding eigenvectors $\mathbf{w_1}$ and $\mathbf{w_2}$ are also orthogonal. Thus if an eigenvalue of $B_{even}$ has multiplicity 2, there are two orthogonal even eigenvectors of $B$ with that eigenvalue.

Now consider an eigenvector $\mathbf{u}$ of $B_{odd}$. As before, one can construct an eigenvector $\mathbf{v}$ of $B'$: for $n_f + n_r + 1 \leq i \leq n$ let $v_i = u_i$; let $v_i = 0$ otherwise. Multiplication by the matrix $T$ again transforms $\mathbf{v}$ into an eigenvector $\mathbf{w}$ of $B$:

$$\mathbf{w} = T\mathbf{v} = \begin{bmatrix} I_{n_f} & 0 & 0 \\ 0 & \frac{1}{\sqrt{(2)}}I_{n_r} & \frac{1}{\sqrt{(2)}}I_{n_r} \\ 0 & \frac{1}{\sqrt{(2)}}I_{n_r} & \frac{-1}{\sqrt{(2)}}I_{n_r} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \mathbf{v}_{\phi(r)} \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{(2)}}\mathbf{v}_{\phi(r)} \\ \frac{-1}{\sqrt{(2)}}\mathbf{v}_{\phi(r)} \end{bmatrix}$$

This is clearly an odd vector. Since $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ all have the same eigenvector $\lambda$, the claim about eigenvalues of $B_{odd}$ corresponding to odd eigenvalues of $B$ holds. It is easy to show that if two eigenvectors $\mathbf{u_1}$ and $\mathbf{u_2}$ of $B_{odd}$ are orthogonal, then the corresponding eigenvectors $\mathbf{w_1}$ and $\mathbf{w_2}$ are also orthogonal. Thus of an eigenvalue of $B_{odd}$ has multiplicity 2, there are two orthogonal odd eigenvectors of $B$ with that eigenvalue.

Note that if all eigenvectors of $B_{even}$ and $B_{odd}$ are transformed in this way the result is $n$ orthogonal eigenvectors of $B$ (i.e., a full set).

It is easy to construct the components $G_{odd}$ and $G_{even}$ from $G$. I now give the rules for these constructions; they are easy to verify from the matrix entries of $B'$.

$G_{odd}$ is a weighted graph on $V_r$. Start with the subgraph of $G$ induced by $V_r$, with the weight of vertex $v_i$ equal to the weight of $v_i$ in $G$ and the weight of edge $(v_i, v_j)$ equal to its weight in $G$. Adjust the weights according to the following two rules:

- **Degree weight adjustment rule:** For each vertex $v_i \in V_r$, if there is an edge in $G$ from $v_i$ to $v_{\phi(i)}$ (i.e., there is an edge from $v_i$ to its image under the automorphism), then increase the weight of $v_i$ in $G_{odd}$ by $w_{i\phi(i)}$.

- **Edge weight adjustment rule:** For each pair of distinct vertices $v_i,\ v_j \in V_r$ and $i < j$, if there is an edge $(v_i, v_{\phi(j)})$ in $G$ (i.e., if there is an edge from $v_i$ in the representative set to the image of $v_j$ outside of the representative set), add an edge $(v_i, v_j)$ with weight $-w_{i,\phi(j)}$ to $G_{odd}$ (if there is already an edge $(v_i, v_j)$ in $G_{odd}$, subtract $w_{i,\phi(j)}$ from its weight). To see that this rule is well-formed with respect to permutations of the vertex numberings, note that since $\phi$ is an automorphism of order 2, the existence of an edge $(v_i, v_{\phi(j)})$ in $G$ implies the existence of an edge $(v_{\phi(i)}, v_j)$ in $E(G)$ with the same weight. That is, for any $i < j$

$$w_{ij}^{G_{odd}} \;=\; w_{ij}^{G} - w_{i\phi(j)}^{G} \;=\; w_{ji}^{G} - w_{j\phi(i)}^{G},$$

  where the superscripts on the weights indicate the graph for which they are defined. Thus, renumbering the vertices in a way that assigns $v_j$ an index less than the new index of $v_i$ does not affect the weight of the corresponding edge in $G_{odd}$.

Delete any zero-weight edges from $G_{odd}$.

$G_{even}$ is a weighted graph on $V_r \cup V_f$. Start with the subgraph of $G$ induced by $V_r \cup V_f$, with the weight of vertex $v_i \in V_r \cup V_f$ equal to its degree in $G$ and the weight of each edge $(v_i, v_j)$ equal to its weight in $G$. Adjust the weights according to the following three rules:

- **Degree weight adjustment rule:** For each vertex $v_i \in V_r$, if there is an edge in $G$ from $v_i$ to $v_{\phi(i)}$ (i.e., there is an edge from $v_i$ to its image under the automorphism), then decrease the weight of $v_i$ in $G_{even}$ by $w_{i\phi(i)}$.

- **$V_r$-to-$V_f$ Edge weight adjustment rule:** For each edge $(v_i, v_j)$ in $G_{even}$ such that $v_i \in V_r$ and $v_j \in V_f$ (i.e., for each edge between the fixed vertices and the representative vertices), multiply its weight by $\sqrt{2}$.

- **$V_r$-to-$V_r$ Edge weight adjustment rule:** For each pair of distinct vertices $v_i,\ v_j \in V_r$ and $i < j$, if there is an edge $(v_i, v_{\phi(j)})$ in $G$ (i.e., if there is an edge from $v_i$ in the representative set to the image of $v_j$ outside of the representative set), add an edge $(v_i, v_j)$ with weight $w_{i,\phi(j)}$ to $G_{even}$ (if there is already an edge $(v_i, v_j)$ in $G_{even}$, add $w_{i,\phi(j)}$ to its weight). I leave it to the reader to check that this rule is well-formed with respect to permutations of the vertex numberings; the argument is the same as for the edge weight adjustment rule in the odd case.

Delete any zero-weight edges from $G_{even}$.

Since

$$|V(G_{odd})| + |V(G_{even})| = |V_f| + 2|V_r| = |V_f| + |V_m| = |V(G)|,$$

the claim about the combined size of the two graphs holds.
□

It is useful to consider an example of even-odd graph decomposition before proceeding. I demonstrate one way in which a complete binary tree of three levels can be decomposed. The initial graph is shown in Figure 2.3.1 below. The first automorphism maps leaves with the same parent

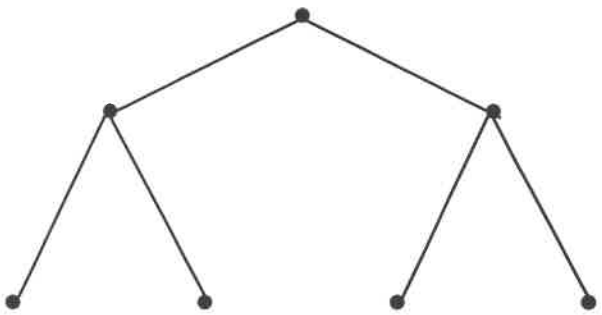


Figure 2.3: The Initial Graph

to each other. Applying the rules given above for constructing the odd and even components gives the result shown in Figure 2.3.1. The odd component is fully decomposed; one can apply one more step of decomposition to the even component using the automorphism that maps the corresponding vertices at the lowest two levels to each other. The result is shown in Figure 2.3.1. This example is also useful in arguments about bounding the smallest nonzero eigenvalue of complete binary trees below.

The following technical lemmas about the eigenvalues and eigenvectors of weighted path graphs are useful in subsequent results.

**Lemma 2.3.5 [Zero Entries of Path Graph Eigenvectors Lemma]** *Let $B$ be the standard matrix representation of a weighted path graph $G$ on $n$ vertices. For any vector $\mathbf{x}$ such that $B\mathbf{x} = \lambda\mathbf{x}$ for some real $\lambda$, $x_n = 0$ implies $\mathbf{x} = 0$. Likewise, $x_1 = 0$ implies $\mathbf{x} = 0$. If there are two consecutive elements $x_i$ and $x_{i+1}$ that are both zero, then $\mathbf{x} = 0$.*

**Proof:** The first result is proved by induction. The base case is for a $2 \times 2$ matrix with diagonal entries $b_{11}$ and $b_{22}$, and off-diagonal entries $b_{12} = b_{21} = -c$. Let $\mathbf{x}$ and $\lambda$ be as specified by the lemma statement, and assume that $x_2 = 0$. The second element of the vector resulting from multiplying $B\mathbf{x} = \lambda\mathbf{x}$ is $-c \cdot x_1 = \lambda x_2 = 0$. Since $c \neq 0$ by definition ($G$ is a weighted path graph), it must be the case $x_1 = 0$, which implies that $\mathbf{x} = 0$.

For the induction step, assume that the result holds for all $i \leq k$, and consider the standard matrix representation of a weighted path graph on $k+1$ vertices. Let the weight of the edge from $v_k$ to $v_{k+1}$ be $c$. Let $\mathbf{x}$ and $\lambda$ be as stated, and assume that $x_{k+1} = 0$. The $k + 1^{\text{st}}$ entry of $B\mathbf{x}$ is $-c \cdot x_k = \lambda x_{k+1} = 0$. Thus $x_k = 0$. Let $\mathbf{x}'$ be the subvector of $\mathbf{x}$ consisting of the first $k$ entries. Note that with $x_{k+1} = 0$ it is the case that $\mathbf{x}'$ and $\lambda$ meet the lemma conditions for the principle leading minor $B_k$ of $B$, and that $x'_k = 0$. But the leading principle minor $B_k$ is the standard matrix representation for the weighted path graph derived from $G$ by deleting the last edge and vertex.

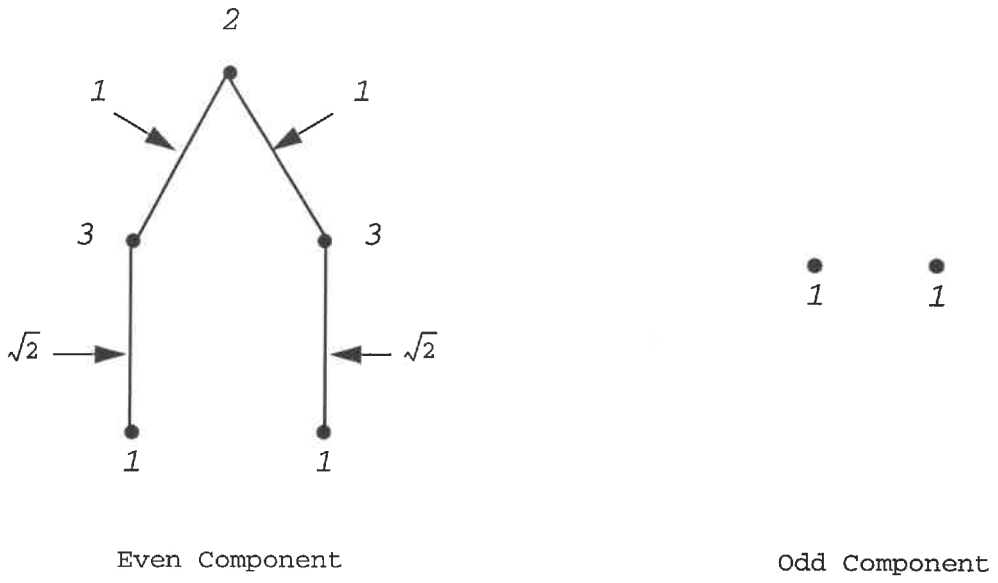



Figure 2.4: After the First Decomposition Step

Thus, by the induction hypothesis $\mathbf{x}'$ must be 0; because $x_{k+1} = 0$ this implies that $\mathbf{x} = 0$

A symmetric argument implies the result for $x_1 = 0$.

Again let $B$ be the standard matrix representation of a weighted path graph $G$. Let $\mathbf{x}$ be a vector meeting the lemma conditions for $\lambda$, and assume that $\mathbf{x}$ has two consecutive zero elements $x_i$ and $x_{i+1}$. If either $i = 1$ or $i + 1 = n$, $\mathbf{x} = 0$ by the previous argument. Otherwise, $x_{i+1} = 0$ implies that the first $i$ elements of $\mathbf{x}$ and $\lambda$ meet the lemma conditions for the leading principle minor $B_i$ of $B$. Note that $B_i$ is the standard matrix representation for some weighted path graph. Thus by the previous result the first $i$ entries of $\mathbf{x}$ are zero. By a symmetric argument for the trailing principle minor, the last $n - i$ entries must also be zero, which gives $\mathbf{x} = 0$.

□

Since eigenvectors are by definition not equal to the zero vector, the theorem above implies that for eigenvectors of the standard matrix representation $B$ of any weighted path graph, neither the first nor the last entry is zero. Likewise, such an eigenvector cannot have two consecutive zero entries. These facts can be used to give a simple proof of the following lemma (for a different proof, see e.g. pp. 910-911 of [YG73]).

**Lemma 2.3.6** *All eigenvalues of the standard matrix representation $B$ of a weighted path graph $G$ on $n$ vertices are simple (i.e., have multiplicity one).*

**Proof:** Let $\mathbf{u}$ and $\mathbf{u}'$ be any two eigenvectors of $B$ for the eigenvalue $\lambda$. By the Zero Entries of Path Graph Eigenvectors Lemma (Lemma 2.3.5), $u_n \neq 0$ and $u'_n \neq 0$. Let $\alpha$ be $u'_n/u_n$; $\alpha$ is non-zero and real. Then $B(\alpha\mathbf{u} - \mathbf{u}') = \lambda(\alpha\mathbf{u} - \mathbf{u}')$. But the $n^{\text{th}}$ element of $(\alpha\mathbf{u} - \mathbf{u}')$ is 0, so by Lemma 2.3.5, it must be the case that $\alpha\mathbf{u} = \mathbf{u}'$, so $\mathbf{u}$ must be a scalar multiple of $\mathbf{u}'$; it is not a

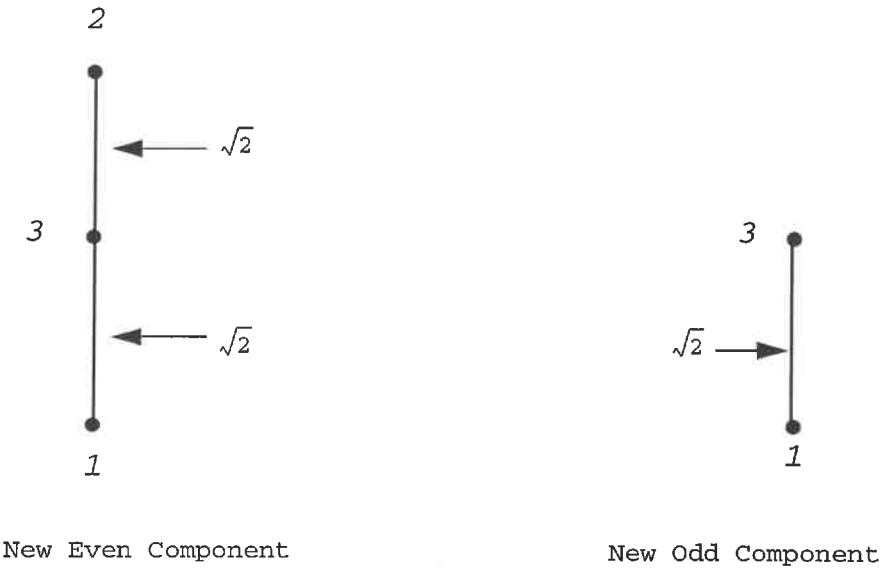



Figure 2.5: Result of Decomposing Odd Component

distinct eigenvector.
□

A path graph on $n$ vertices has exactly one automorphism of order two: $\phi(i) = n - i + 1$. Thus one can talk about odd and even eigenvectors of a path graph without ambiguity; they are always with respect to this automorphism.

**Lemma 2.3.7** *The eigenvector* $\mathbf{u}_2$ *corresponding to the second smallest eigenvalue* $\lambda_2$ *of the Laplacian* $B$ *of a path graph* $G$ *on* $n$ *vertices is odd.*

**Proof:** By Lemma 2.3.6, $\mathbf{u}_2$ is simple, so by Corollary 2.3.3, $\mathbf{u}_2$ must be either even or odd. Assume that it is even. I show this leads to a contradiction.

Recall the characterization

$$\lambda_2 = \min_{\mathbf{x} \perp \vec{1}} \frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}}.$$

There are two cases to keep track of: $n$ is odd, and $n$ is even. If $n$ is odd, there is a single center vertex $v_{\lceil \frac{n}{2} \rceil}$ (index the vertices along the path from 1 to $n$). If $n$ is even, there are two center vertices with indices $\frac{n}{2}$ and $\frac{n}{2} + 1$; since $\mathbf{u}_2$ is assumed to be even, their entries in $\mathbf{u}_2$ are equal. Thus, by Lemma 2.3.5, if $n$ is even the eigenvector entries corresponding to the center vertices are non-zero. If $n$ is odd, $\mathbf{u}_2$ is even, and the eigenvector entry for the center vertex is 0, then it is easy to check that changing the signs of all eigenvector entries with index less than the center index gives an odd eigenvector with eigenvalue $\lambda_2$, which contradicts the simplicity of $\lambda_2$. Thus, the assumption that $\mathbf{u}_2$ is even implies that the eigenvector entries corresponding to the center vertex or vertices must be non-zero. Let this value be $c$.

Now consider the vector $\mathbf{x} = (-c) \cdot \vec{1} + \mathbf{u}_2$. Recall that $\mathbf{u}_2$ is orthogonal to $\vec{1}$. It is easy to see that $\mathbf{x}$ is even, and since $c \neq 0$,

$$\frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{c^2 \cdot \left(\vec{1}^T A \vec{1}\right) + \mathbf{u}_2^T A \mathbf{u}_2}{\left((-c) \cdot \vec{1} + \mathbf{u}_2\right)^T \left((-c) \cdot \vec{1} + \mathbf{u}_2\right)} = \frac{\mathbf{u}_2^T A \mathbf{u}_2}{c^2 n + \mathbf{u}_2^T \mathbf{u}_2} < \frac{\mathbf{u}_2^T A \mathbf{u}_2}{\mathbf{u}_2^T \mathbf{u}_2}.$$

However, the entries of $\mathbf{x}$ corresponding to the center vertex or vertices are 0, so as above, one can create an odd vector $\mathbf{y}$ such that

$$\frac{\mathbf{y}^T A \mathbf{y}}{\mathbf{y}^T \mathbf{y}} = \frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$$

as follows: set $y_i = x_i$, $i < \frac{n}{2}$ and $y_i = -x_i$, $i > \frac{n}{2}$. $\mathbf{y}$ is orthogonal to $\vec{1}$, so it meets the criteria for the characterization of $\lambda_2$, so the assumption that $\mathbf{u}_2$ is even gives $\lambda_2 < \lambda_2$, a contradiction. □

### 2.3.2 Bounds on $\lambda_2$ for Trees and Double Trees

I now use the results from the preceding section to prove some upper and lower bounds on $\lambda_2$ for the Laplacian of the complete binary tree and for the Laplacian of a graph I call the **double tree**. A double tree is two complete binary trees of $k$ levels for some $k > 0$ connected by an edge between their respective roots. The bounds developed below are useful in Section 2.5.

To bound the size of $\lambda_2$ of a complete binary tree, I first apply the Even-Odd Decomposition Theorem (Theorem 2.3.4) $k - 1$ times to show that the eigenvalues of a balanced binary tree can be computed from a few simple types of weighted graphs. I then bound the eigenvalues for these types of graphs using the interlacing theorems and matrix perturbation techniques. The result is a proof that for $k \geq 3$, a complete binary tree on $k$ levels with $n = 2^k - 1$ vertices, $\lambda_2 = \Theta(\frac{1}{n})$. More specifically, $\frac{1}{n} < \lambda_2 < \frac{2}{n}$.

For the following argument, assume that the initial graph is a complete binary tree of $k > 2$ levels. As above, $n = 2^k - 1$.

Start by applying the Even-Odd Graph Decomposition Theorem with respect to the automorphisms that each map one pair of sibling leaves to each other. There are $2^{k-1}$ leaves and $2^{k-2}$ such automorphisms. For the odd eigenvalues, $\lambda = 1$ for each of the $2^{k-2}$ single-vertex graphs that comprise the odd components. The result for $k = 3$ was shown above in Figure 2.3.1.

To help in understanding the structure of the resulting even component, I introduce the following terminology: for $i \geq 3$, the $i^{\text{th}}$ **odd collapsed tree graph** is a weighted graph on $i - 1$ vertices, with every edge having weight $\sqrt{2}$, vertex $v_1$ having weight 1, and all other vertices having weight 3 (the odd collapsed tree graph for $i = 5$ is shown in Figure 2.3.2 below). With this terminology, $G_{even}$ can be described in the following way: the even component after the first decomposition step has the structure of a complete binary tree of $k - 1$ levels, except that the leaves of the tree are replaced by the odd collapsed tree graph for $i = 3$; the vertices at level $k - 2$ are connected by edges of weight 1 to the vertices $v_2$ of the odd collapsed tree graphs. This is illustrated for $k = 3$ in Figure 2.3.1 above.

Now repeat the process for the $2^{k-3}$ automorphisms that map neighboring pairs of the odd collapsed tree graphs to each other. The odd eigenvalues from these decompositions are the

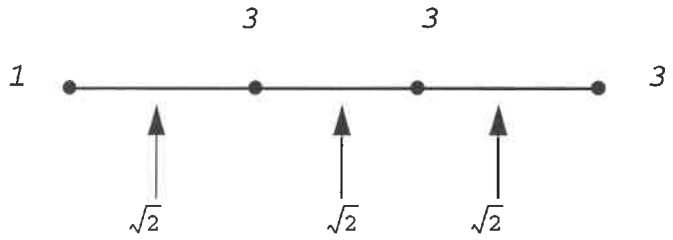



Figure 2.6: Odd Collapsed Tree Graph, $i = 5$

eigenvalues of the odd collapsed tree graph for $i = 3$, each occurring $2^{k-3}$ times; the even eigenvalues come from a new even component, a weighted graph that looks like a complete binary tree with $k - 2$ levels, except that the leaves are replaced by odd collapsed tree graphs for $i = 4$. Once again, each odd collapsed tree is connected to its parent by an edge of weight 1 from vertex $v_3$ of the odd collapsed tree graph.

Continue this process; at the $j^{\text{th}}$ series of reductions the odd eigenvalues are those of the odd collapsed tree graph for $i = j + 1$; the even component is a weighted graph that looks like a complete binary tree with $k - j$ levels, except that the leaves are replaced by odd collapsed tree graphs for $i = j + 2$.

At the last decomposition step, start with a weighted graph that looks like a complete binary tree with 2 levels, except that the leaves have been replaced by odd collapsed tree graphs for $i = k$. Apply the Even-Odd Decomposition Theorem one last time; the eigenvalues for this graph are those of the odd collapsed tree graph for $i = k$ plus the eigenvalues for a graph on $k$ vertices that consists of the odd collapsed tree graph for $i = k$ plus vertex $v_k$ with weight 2, and edge $(v_{k-1}, v_k)$ with weight $\sqrt{2}$ (I refer to this latter graph as the **even collapsed tree graph** for $i = k$; the even collapsed tree graph for $i = 5$ is shown in Figure 2.3.2 below). Recall that the original tree's eigenvalues that are not represented by either of these final components are either 1 (from the first set of decompositions) or eigenvalues for the odd collapsed tree graphs for $i$, $3 \leq i < k$.

Let $\mu^{(k)}$ be the smallest eigenvalue of the odd collapsed tree graph for $i = k$. I claim that $\lambda_2$ for the complete binary tree on $k$ levels is equal to $\mu^{(k)}$. $\mu^{(k)}$ is less than $\mu^{(i)}$ for any $i < k$ by the First Interlacing Theorem. The argument below shows that $\mu^{(k)} < 1$ for $k \geq 3$. It is easy to show that the standard matrix representation for the even collapsed tree graph for $i = k$ is singular, so its smallest eigenvalue is 0. Thus, an application of the First Interlacing Theorem shows that the smallest eigenvalue of the odd collapsed tree graph for $i = k$ is less than or equal to the first non-zero eigenvalue of the even collapsed tree graph for $i = k$ (the standard matrix representation of the former is a leading principle submatrix of the latter).

Bounds on $\mu^{(k)}$ are computed using matrix perturbation techniques. Note that the standard matrix representation for the odd collapsed tree is nonsingular, so the smallest eigenvalue is the one of interest.

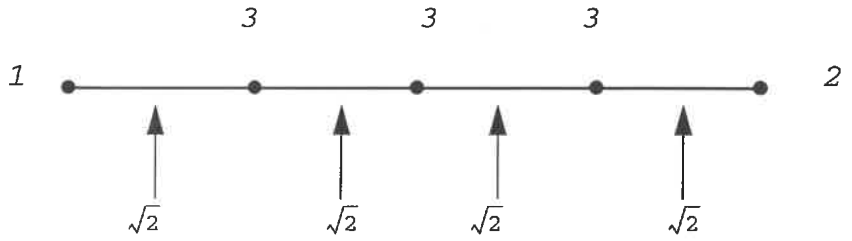



Figure 2.7: Even Collapsed Tree Graph, $i = 5$

The rest of this argument uses two matrices. The first, $B$, is the standard matrix representation for the odd collapsed tree graph for $i = k$. The determinant of $B$ is 1 (this is easily proved by induction on $k$). The vertex ordering is as given in the description of the graph above. The second matrix, $B'$, has

$$b'_{11} = b_{11} - \frac{1}{2^{k-1}-1} = 1 - \frac{1}{2^{k-1}-1} = \frac{n-3}{n-1},$$

and all other entries are the same as for $B$. The determinant of $B'$ is 0 (again, easily shown via induction).

Let $\mathbf{v}$ be the eigenvector of $B'$ corresponding to $\lambda = 0$ (assume without loss of generality that it is scaled to unit length). Then

$$\mathbf{v}^T B' \mathbf{v} = \mathbf{v}^T B \mathbf{v} - \frac{v_1^2}{2^{k-1}-1} = 0.$$

By the Courant-Fischer Minimax Theorem (see, e.g., [GL89]), $\frac{v_1^2}{2^{k-1}-1}$ is an upper bound on $\mu^{(k)} = \lambda_2(B)$. The next step is to show that for $k \geq 3$, this upper bound is strictly less than $\frac{2}{n}$. Note that this immediately implies that $\mu^{(k)}$ is less than 1.

Because $2^{k-1} - 1 = \frac{n-1}{2}$, the inequality $\frac{v_1^2}{2^{k-1}-1} < \frac{2}{n}$ holds if $v_1^2 < \frac{n-1}{n}$. Assume that $v_1^2 \geq \frac{n-1}{n}$; I show that this assumption contradicts the fact that $\mathbf{v}$ is unit length. Since $B'\mathbf{v} = 0$, the definition of $B'$ implies that

$$b_{11}v_1 - \sqrt{2}v_2 = \frac{n-3}{n-1}v_1 - \sqrt{2}v_2 = 0 \;\rightarrow\; v_2 = \frac{1}{\sqrt{2}}\left(\frac{n-3}{n-1}\right)v_1,$$

and by the assumption on $v_1^2$,

$$v_2^2 = \frac{1}{2}\left(\frac{n-3}{n-1}\right)^2 v_1^2 \geq \frac{1}{2n}\frac{(n-3)^2}{n-1}.$$

Since $\mathbf{v}$ is unit length, the assumption yields

$$1 = \mathbf{v}^T\mathbf{v} = \sum_{i=1}^{k-1} v_i^2 \geq v_1^2 + v_2^2 = \frac{1}{n}\left(n - 1 + \frac{1}{2}\frac{(n-3)^2}{n-1}\right) > 1,$$

a contradiction (the last inequality holds because if $k \geq 3$, then $n \geq 7$, and thus $\frac{(n-3)^2}{n-1} > 2$).

To get a lower bound on $\mu^{(k)}$, let $\mathbf{u}$ be the unit-length vector such that $\mathbf{u}^T B\mathbf{u}$ is minimized (i.e., $\mathbf{u}^T B\mathbf{u} = \mu^{(k)}$):

$$\mathbf{u}^T B'\mathbf{u} = \mathbf{u}^T B\mathbf{u} - \frac{u_1^2}{2^{k-1}-1} = \mu^{(k)} - \frac{u_1^2}{2^{k-1}-1} \geq 0.$$

The last inequality holds by a version of the Second Interlacing Property with $\alpha < 0$; $B'$ has a zero eigenvalue, and all its other eigenvalues are greater than or equal to the (positive) eigenvalues of $B$. $B'$ is thus positive semidefinite.

The above equation implies that if $u_1 > \frac{1}{\sqrt{2}}$, then $\mu^{(k)} > \frac{1}{2^{k-2}} > \frac{1}{n}$. Since $B$ is tridiagonal, this is easily done by using $B$ and $\mu^{(k)}$ to generate a recurrence for the entries of $\mathbf{u}$.

Consider the first entry of $B\mathbf{u}$. Using the fact that $\mathbf{u}$ is an eigenvector and the structure of $B$, it is clear that $u_1 - \sqrt{2}u_2 = \mu^{(k)}u_1$. Since $1 > \mu^{(k)} > 0$, this yields $u_2 < \frac{u_1}{\sqrt{2}}$. This provides the base case for a proof by induction that $u_{i+1} < \frac{u_i}{\sqrt{2}}$. The details are left to the reader; the previous inequality serves as the induction hypothesis. The further entries of the product $B\mathbf{u}$ are expressed by the equations

$$3u_i - \sqrt{2}(u_{i-1} + u_{i+1}) = \mu^{(k)}u_i,$$

which can be used to prove the desired result for $u_{i+1}$.

Since $\mathbf{u}$ is unit length,

$$1 = \mathbf{u}^T\mathbf{u} = \sum_{i=1}^{k-1} u_i^2 < \sum_{i=1}^{k-1}\left(\frac{1}{\sqrt{2}}\right)^{2(i-1)} u_1^2 = u_1^2 \sum_{i=1}^{k-1}\left(\frac{1}{2}\right)^{(i-1)} < 2u_1^2,$$

which gives the desired result that $u_1 > \frac{1}{\sqrt{2}}$.

The preceding argument proves the following lemma:

**Lemma 2.3.8** *For a complete balanced binary tree on $k \geq 3$ levels and $n = 2^k - 1$ vertices, $\frac{1}{n} < \lambda_2 < \frac{2}{n}$.*

For double trees where each of the component trees has $k$ levels and $n = 2^{k+1} - 2$ vertices, the following lemma applies:

**Lemma 2.3.9** *For a double tree on $n \geq 14$ vertices, $\frac{1}{n} < \lambda_2 < \frac{4}{n}$.*

**Proof:** The proof uses a number of facts from the proof of the $\lambda_2$ bounds for complete binary trees given above.

Start by noting that one can apply the Even-Odd Decomposition Theorem starting from the leaves of the trees as done above. The odd eigenvalues determined at each step are the same as for the complete binary tree.

The last decomposition step (the one that divides between the two roots) results in two components: the even component $G_e$, which is the same as the even component for a tree of $k$

levels, and the odd component $G_o$, which is like the $k+1^{st}$ odd collapsed tree graph, except that the weight of vertex $k$ is 4 rather than 3. Again, the standard matrix representation is singular for $G_e$ and non-singular for $G_o$; an application of the Second Interlacing Theorem thus implies that $\lambda_2$ for the double tree is the smallest eigenvalue for $G_o$.

Next, note that the standard matrix representation of the odd collapsed tree for $i = k$ is a principle submatrix of the standard matrix representation for $G_o$. The First Interlacing Theorem and the analysis of the odd collapsed tree above immediately imply that $\lambda_2 < \frac{2}{2^k-1}$; since the double tree has $n = 2^{k+1} - 2$ vertices, this implies that $\lambda_2 < \frac{4}{n}$.

Finally, note that $G_o$ is the odd collapsed tree for $i = k+1$ with 1 added to the weight of vertex $k$. One can thus apply the Second Interlacing Theorem and the results for the odd collapsed tree above (recall that in the proof it was shown that $\mu^{(k)} > \frac{1}{2^k-2}$) to show that $\lambda_2 \geq \mu^{(k+1)} > \frac{1}{2^{k+1}-2}$; since the double tree has $n = 2^{k+1} - 2$ vertices, this implies that $\lambda_2 > \frac{1}{n}$.

□

## 2.4 A Bad Family of Bounded-Degree Planar Graphs for Spectral Bisection

In this section I present a family of bounded-degree planar graphs that have constant-size separators. However, the separators produced by spectral bisection have size $\Theta(n)$ for both edge and vertex separators. Since there are algorithms that produce $O(\sqrt{n})$ vertex separators for planar graphs, and hence $O(\sqrt{n})$ edge separators for bounded-degree planar graphs, spectral bisection performs poorly on these graphs relative to other algorithms.

The family of graphs is parameterized on the positive integers. $G_k$ consists of two path graphs, each on $3k$ vertices, with a set of edges between the two paths as follows: label the vertices of one path from 1 to $3k$ in order (the **upper path**), and label the other path from $3k+1$ to $6k$ in order (the **lower path**). For $1 \leq i \leq k$ there is an edge between vertices $2k+i$ and $5k+i$. This was shown in Figure 2.1 in the Introduction. It is obvious that $G_k$ is planar for any $k$, and that the maximum degree of any vertex is 3.

Note that the graph has the approximate shape of a cockroach, with the section containing edges between the upper and lower paths being the body, and the other sections of the paths being antennae. This terminology allows easy references to parts of the graph.

$G_k$ has one automorphism of order 2 that maps the vertices of the upper path to the vertices of the lower path and vice versa. For the rest of this section, the terms "odd vector" and "even vector" are used with respect to this automorphism. Thus, an even vector $\mathbf{x}$ has $x_i = x_{3k+i}$ for all $i$ in the range $1 \leq i \leq 3k$; an odd vector $\mathbf{y}$ has $y_i = -y_{3k+i}$ for all $i$, $1 \leq i \leq 3k$.

I can now discuss the structure of the eigenvectors of the Laplacian of $G_k$.

**Lemma 2.4.1** *Any eigenvector $\mathbf{u}_i$ with eigenvalue $\lambda_i$ of the Laplacian $B_k$ of $G_k$ can be expressed in terms of linear combinations of even and odd eigenvectors with eigenvalue $\lambda_i$. In particular, $\mathbf{u}_i$ can be expressed as a linear combination of:*

- *an even eigenvector of $B_k$ in which the values associated with the upper path are the same as for the eigenvector with eigenvalue $\lambda_i$ (if it exists) of a path graph on $3k$ vertices, and*

- *an odd eigenvector of $B_k$ in which the values associated with the upper path are the same as for the eigenvector with eigenvalue $\lambda_i$ (if it exists) of a weighted graph that consists of a path graph on $3k$ vertices for which the vertex weights of $v_{2k+1}$ through $v_{3k}$ have been increased by 2.*

**Proof:** The first claim follows by the Even-Odd Eigenvector Theorem applied with respect to the automorphism that maps the vertices of the upper path to the vertices of the lower path and vice versa.

The claim about the specific structure of the $\mathbf{u}_i$'s follows by a direct application of the construction used in the proof of the Even-Odd Graph Decomposition Theorem with respect to the same automorphism.

□

I now give the proof that spectral bisection gives bad separators for the family of graphs $G_k$.

**Theorem 2.4.2** *Spectral bisection produces $\Theta(n)$ edge and vertex separators for $G_k$ for any $k$.*

**Proof:** The first step is to show that $\mathbf{u}_2$ is odd. Intuitively, this implies that the spectral method splits the graph into the upper path and the lower path.

Recall that $\lambda_2 = \min_{\mathbf{x} \perp \vec{1}} \frac{\mathbf{x}^T B_k \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$. I construct an odd vector $\mathbf{x}$ such that the quotient $\frac{\mathbf{x}^T B_k \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$ is less than $\frac{\mathbf{y}^T B_k \mathbf{y}}{\mathbf{y}^T \mathbf{y}}$ for any even eigenvector $\mathbf{y}$ orthogonal to $\vec{1}$ ($\vec{1}$ is the smallest even eigenvector). This requires a proof that $\frac{\mathbf{x}^T B_k \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$ is less than the second smallest even eigenvalue. From Lemma 2.4.1 above, the second smallest even eigenvalue of $B_k$ is the same as the second smallest eigenvalue of the Laplacian of a path graph on $3k$ vertices; it is well-known that this value is $4 \sin^2(\frac{\pi}{6k})$ (see for example [Moh88]).

Let $\mathbf{z}$ be the eigenvector corresponding to the second smallest eigenvalue $\mu_2$ for the Laplacian $B$ of the path graph $G$ on $4k$ vertices ($\mu_2 = 4 \sin^2(\frac{\pi}{8k})$). Construct $\mathbf{x}$ as follows:

$$x_i = \begin{cases} z_i & 1 \leq i \leq 2k, \\ z_{7k-i+1} & 3k+1 \leq i \leq 5k, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

That is, assign the first $2k$ values from the path $G$ to the upper antenna of the roach, working in the direction towards the body, and assign the last $2k$ entries from $G$ to the lower antenna, working from the body outward. Since $\mathbf{z}$ and $\mathbf{x}$ have the same set of non-zero entries, $\mathbf{x}^T\mathbf{x} = \mathbf{z}^T\mathbf{z}$. Likewise, since $\mathbf{z}$ is orthogonal to the "all-ones" vector, so is $\mathbf{x}$.

To see that $\mathbf{x}^T B_k \mathbf{x} < \mathbf{z}^T B \mathbf{z}$, recall (2.1) from Section 2.3: for Laplacian $A$ and vector $\mathbf{y}$,

$$\mathbf{y}^T A \mathbf{y} = \sum_{(v_i, v_j) \in E} (y_i - y_j)^2.$$

For every edge in $G$ except one, there is an edge in $G_k$ that contributes the same value to this sum. The one exception is the edge $(v_{2k}, v_{2k+1})$. Since $\mathbf{z}$ is an odd vector by Lemma 2.3.7, and since $\mathbf{z}$ has an even number of entries, $z_{2k} = -z_{2k+1}$. By the Zero Entries of Path Graph Eigenvectors Lemma (Lemma 2.3.5), it is not possible for both $z_{2k}$ and $z_{2k+1}$ to be zero, so $z_{2k}$ is equal to some non-zero value $c$, and this edge contributes $4c^2$ to the value of $\mathbf{z}^T B \mathbf{z}$. On the other hand, there are

two edges in $G_k$ that contribute non-zero values and that do not have corresponding edges in $G$: $(v_{2k}, v_{2k+1})$ and $(v_{5k}, v_{5k+1})$. Each of these edges contributes $c^2$ to $\mathbf{x}^T B_k \mathbf{x}$. Thus

$$\mathbf{x}^T B_k \mathbf{x} = \mathbf{z}^T B \mathbf{z} - 4c^2 + 2c^2 = \mathbf{z}^T B \mathbf{z} - 2c^2 < \mathbf{z}^T B \mathbf{z}.$$

Since $\mathbf{x}^T \mathbf{x} = \mathbf{z}^T \mathbf{z}$,

$$\lambda_2(G_k) \leq \frac{\mathbf{x}^T B_k \mathbf{x}}{\mathbf{x}^T \mathbf{x}} < \frac{\mathbf{z}^T B \mathbf{z}}{\mathbf{z}^T \mathbf{z}} = 4\sin^2(\frac{\pi}{8k}) < 4\sin^2(\frac{\pi}{6k}).$$

That is, the second smallest eigenvalue of $B_k$ is less than any non-zero even eigenvalue, and is thus odd by the Even-Odd Eigenvector Theorem (Theorem 2.3.2).

There are a few details to finish off. In particular, it is necessary to show that there are not too many zero entries in $\mathbf{u}_2$ (spectral bisection as defined in this paper does not separate vertices with the same value). Since $\mathbf{u}_2$ is an odd vector and since the odd component of $G_k$ is a weighted path graph, Lemmas 2.3.5 (the Zero Entries of Path Graph Eigenvectors Lemma) and 2.4.1 imply that $\mathbf{u}_2$ cannot have consecutive zeros, and the values corresponding to vertices $3k$ and $6k$ are non-zero. Thus the edge separator generated by spectral bisection must cut at least half the edges between the upper and lower paths; since none of these edges share an endpoint, the cover used in generating the vertex separator must include at least this number of vertices.
□

## 2.5 A Bad Family of Graphs for the "Best Threshold Cut" Algorithm

While the roach graph defeats spectral bisection, the second smallest eigenvector can still be used to find a small separator using the "best threshold cut" algorithm. In particular, Theorem 2.3.1 implies that considering all threshold cuts induced by $\mathbf{u}_2$ produces a constant-size cut: If $q_{min}$ is the minimum cut quotient for these cuts, then

$$q_{min} \leq \sqrt{\lambda_2(2\Delta - \lambda_2)} \leq \frac{\sqrt{6}\pi}{4k},$$

which implies $q_{min}$ is $O(\frac{1}{n})$. Since the denominator of $q_{min}$ is less than or equal to $\frac{n}{2}$, the number of edges in this cut must be bounded by a constant.

In this section I show that there is a family of graphs for which the "best threshold cut" algorithm does poorly. Recall that Section 2.3.2 defined a double tree as a graph that consists of two complete binary trees of $k$ levels for some $k > 0$, connected by an edge between their respective roots. The **tree-cross-path graph** consists of the crossproduct of a double tree on $p_1$ vertices and a path graph on $p_2$ vertices. I show below that there are tree-cross-path graphs that defeat the "best threshold cut" algorithm. To do so, I use the bounds from Section 2.3.2 on $\lambda_2$ for trees and double trees.

I formally state the result for this section as follows:

**Theorem 2.5.1** *There exists a graph $G$ for which the "best threshold cut" algorithm finds a separator $S$ such that the cut quotient for $S$ is bigger than $i(G)$ by a factor as large (to within a constant) as allowed by the bounds from Theorem 2.3.1.*

**Proof:** Let $G$ be the tree-cross-path graph that is the crossproduct of a double tree of size $p$ and a path of length $cp^{\frac{1}{2}}$ for some $c$ in the range $3.5 \leq c < 4$. To insure that the double tree and the path have integer sizes, restrict $p$ to integers of the form $2^k - 2$ for $k > 2$. Then choose $c$ in the range specified such that $cp^{\frac{1}{2}}$ is an integer (the choice of $p$ insures there is an integer in this range).

Recall that the eigenvalues of a graph crossproduct are all pairwise sums of the eigenvalues from the graphs used in the crossproduct operation. Let $\nu_2$ be the second smallest eigenvalue of the double tree on $p$ vertices, and let $\mu_2$ be the second smallest eigenvalue for the path on $cp^{\frac{1}{2}}$ vertices. If $\mu_2 < \nu_2$, then $\lambda_2$ for the crossproduct is $\mu_2$ (i.e., $\mu_2$ added to the zero eigenvalue of the double tree). But $\mu_2 = 4\sin^2(\frac{\pi}{2cp^{\frac{1}{2}}})$, and $\nu_2$ is greater than or equal to $\frac{1}{p}$. Therefore it is necessary to show that

$$4\sin^2\left(\frac{\pi}{2cp^{\frac{1}{2}}}\right) < \frac{1}{p}.$$

Reorganizing, simplifying, and noting that $\sin(\theta) < \theta$ for $0 < \theta \leq \frac{\pi}{2}$, it is sufficient to show that

$$\frac{\pi}{2cp^{\frac{1}{2}}} < \frac{1}{2p^{\frac{1}{2}}}, \text{ or } \pi < c.$$

Clearly by the choice of $c$ this inequality holds.

Since path graph eigenvalues are simple (Lemma 2.3.6), the second smallest eigenvalue of $G$ is also simple.

Note that the tree-cross-path graph can be thought of as $cp^{\frac{1}{2}}$ copies of the double tree, each corresponding to one vertex of the path graph. Each vertex in the $i^{\text{th}}$ copy of the double tree is connected by an edge to the corresponding vertex in copies $i-1$ and $i+1$. This description allows one to construct the eigenvector for the second smallest tree-cross-path eigenvalue as follows: Assign each vertex in double tree copy $i$ the value for vertex $i$ in the path graph eigenvector for $\mu_2$. Note that this is the only possible eigenvector since the eigenvalue is simple.

Now consider any copy of the double tree: every vertex in that copy gets the same value in the characteristic valuation. Thus the cut $S$ made by the "best threshold cut" algorithm must separate at least two copies of the double tree, and thus must cut at least $p$ edges. There is a bisection $S^*$ of size $cp^{\frac{1}{2}}$ (cut the edge between the roots in each double tree); because this cut is a bisection, the ratio between the cut quotient $q$ for $S$ and $i(G)$ is at least as large as the ratio between the sizes of these cuts:

$$\frac{q}{i(G)} \geq \frac{|S|}{|S^*|} \geq \frac{p}{cp^{\frac{1}{2}}} = \Omega\left(p^{\frac{1}{2}}\right).$$

From Theorem 2.3.1,

$$\frac{\lambda_2}{2} \leq i(G) \leq q \leq \sqrt{\lambda_2(2\Delta - \lambda_2)}.$$

This plus the fact that the tree-cross-path graph has bounded degree ($\Delta = 5$) implies that

$$\frac{q}{i(G)} \leq \frac{2\sqrt{\lambda_2(2\Delta - \lambda_2)}}{\lambda_2} = O\left(\frac{1}{\sqrt{\lambda_2}}\right) = O\left(p^{\frac{1}{2}}\right).$$

These two bounds imply that, to within a constant factor, the ratio is as large as possible, and the theorem holds.
□

## 2.6 A Bad Family of Graphs for Generalized Spectral Algorithms

The results of the previous section can be extended to more general algorithms that use some number $k$ (where $k$ might depend on $n$) of the eigenvectors corresponding to the $k$ smallest non-zero eigenvalues. In particular, consider algorithms that meet the following restrictions:

- The algorithm computes a value for each vertex using only the eigenvector components for that vertex from $k$ eigenvectors corresponding to the smallest non-zero eigenvalues (for convenience, I refer to these as the $k$ **smallest eigenvectors**). The function computed can be arbitrary as long as its output depends only on these inputs.
- The algorithm partitions the graph by choosing some threshold $t$ and then putting all vertices with values greater than $t$ on one side of the partition, and the rest of the vertices on the other side.
- The algorithm is free to compute the break point $t$ in any way; e.g., checking the separator ratio for all possible breaks and choosing the best one is allowed.

I call such an algorithm **purely spectral**.

The following theorem gives a bound on how well such algorithms do when the number of eigenvectors used is a constant:

**Theorem 2.6.1** *Consider the purely spectral algorithms that use the $k$ smallest eigenvectors for $k$ a fixed constant. Then there exists a family of graphs $\mathcal{G}$ such that $G \in \mathcal{G}$ has a bisection $S^*$ with $|S^*| \geq (k^2 n)^{\frac{1}{3}}$, and such that any purely spectral algorithm using the $k$ smallest eigenvectors produces a separator $S$ for $G$ such that $|S| \geq \left(\frac{|S^*|}{\pi k+1}\right)^2$.*

**Proof:** I show that $\mathcal{G}$ is the set of tree-cross-path graphs that are the crossproducts of double trees of size $p$ (where $p$ is an integer of the form $2^k - 2$ for $k \geq 3$) and paths of length $cp^{\frac{1}{2}}$, where $c$ is a constant chosen such that $\pi k < c \leq \pi k + 1$ and $cp^{\frac{1}{2}}$ is an integer.

Recall that the eigenvalues of a graph crossproduct $G \times H$ are all pairwise sums of the eigenvalues from the graphs $G$ and $H$. Assume for the moment that the $k$ smallest nonzero eigenvalues of any $G \in \mathcal{G}$ are the same as the $k$ smallest nonzero eigenvalues of the path graph used in defining $G$. This clearly holds if these $k$ eigenvalues are smaller than $\lambda_2$ of the double tree; in that case the $k$ smallest non-zero eigenvalues of the crossproduct are these eigenvalues from the path graph added to the zero eigenvalue of the double tree. Since the path graph eigenvalues are simple (Lemma 2.3.6), the corresponding tree-cross-path eigenvalues are also simple.

Note that the tree-cross-path graph can be thought of as $cp^{\frac{1}{2}}$ copies of the double tree, each corresponding to one vertex of the path graph. Each vertex in the $i^{\text{th}}$ copy of the double tree is connected by an edge to the corresponding vertex in copies $i-1$ and $i+1$. This description allows the construction of an eigenvector for each of these $k$ tree-cross-path eigenvalues as follows:

Assign each vertex in double tree copy $i$ the value for vertex $i$ in the path graph eigenvector for this eigenvalue. Note that these are the only possible eigenvectors since these eigenvalues are simple.

The purely spectral algorithm produces a cut $S$ with cut quotient $q$. Recall the assumption about the $k$ smallest eigenvectors and consider any copy of the double tree: since every vertex in that copy gets the same value from each eigenvector, the algorithm assigns the same value to each vertex in this copy. This implies that $S$ must separate at least two copies of the double tree, and thus must cut at least $p$ edges.

There is bisection $S^*$ of size $cp^{\frac{1}{2}}$ (it cuts the edge between the roots in each double tree); because $n = cp^{\frac{3}{2}}$ and $c > k$, it is the case that $|S^*| > k^{\frac{2}{3}} n^{\frac{1}{3}}$. It is obvious that

$$|S| \geq \left(\frac{|S^*|}{c}\right)^2;$$

since $c \leq \pi k + 1$, the claim in the theorem statement holds if the assumption holds.

To prove that the assumption about the form of the $k$ smallest eigenvectors holds for all $G \in \mathcal{G}$, I still need to prove that a path graph on $cp^{\frac{1}{2}}$ vertices has $k$ nonzero eigenvalues smaller than $\lambda_2$ for a double tree on $p$ vertices. Recall that the eigenvalues of a path graph on $l$ vertices are $4\sin^2(\frac{\pi i}{2l})$ for $0 \leq i < l$, and that $\lambda_2$ for a double tree on $p$ vertices is greater than or equal to $\frac{1}{p}$. Therefore it is sufficient to show that

$$4\sin^2\left(\frac{\pi k}{2cp^{\frac{1}{2}}}\right) < \frac{1}{p}.$$

Reorganizing, simplifying, and noting that $\sin(\theta) < \theta$ for $0 < \theta \leq \frac{\pi}{2}$ gives

$$\frac{\pi k}{2cp^{\frac{1}{2}}} < \frac{1}{2p^{\frac{1}{2}}}, \quad \text{or} \quad \pi k < c.$$

Clearly this inequality holds.

□

Note that for the case in which $k$ is constant, the following results apply:

- the cut quotient $q_S$ is no better than the best cut quotient $q_{min}$ produced by considering all threshold cuts for $\mathbf{u}_2$, and
- the gap between $i(G)$ and $q_{min}$ is as large as possible (within a constant factor) with respect to Theorem 2.3.1.

These results can be shown using techniques from the previous section. Thus, $G$ is a graph for which using $k$ eigenvectors does not improve the performance of the "best threshold cut" algorithm.

These results also hold for certain variants of the definition of "purely spectral". For example, Chan, Gilbert, and Teng have proposed using the entries of eigenvectors 2 through $d+1$ of the Laplacian as spatial coordinates for the corresponding vertices of a graph [CGT94]. The graph is then partitioned using a geometric separator algorithm [MTV91],[GMT95]. If this technique is applied (using a fixed $d$) to the counterexample graph used in the proof above, all vertices in a particular copy of the double tree end up with the same coordinates; the geometric algorithm then cuts between copies of the double tree, yielding the same bad cuts as in the proof.

### 2.6.1 Purely Spectral Algorithms that Use More than a Constant Number of Eigenvectors

There are still a number of open questions about the performance of purely spectral algorithms that use more than a constant number of eigenvectors (in particular, how well can such algorithms do if they use all the eigenvectors?). However, just using more than a constant number of eigenvectors is not sufficient to guarantee good separators. In particular, the counterexamples and arguments in the previous sections can be extended to prove the following theorem:

**Theorem 2.6.2** *For sufficiently large $n$ and $0 < \epsilon < \frac{1}{4}$, there exists a bounded-degree graph $G$ on $n$ vertices such that any purely spectral algorithm using the $n^\epsilon$ smallest eigenvectors produces a separator $S$ for $G$ with a cut quotient greater than $i(G)$ by at least a factor of $n^{\left(\frac{1}{4}-\epsilon\right)} - 1$.*

**Proof:** Once again, let $G$ be the tree-cross-path graph. As in the previous two proofs, choose $p_1$ (the double-tree size) and $p_2$ (the path size) such that the smallest $n^\epsilon$ eigenvalues of the crossproduct are the same as the smallest $n^\epsilon$ eigenvalues of the path graph. Once again, a purely spectral algorithm separates two adjacent double trees, while the edges between the roots of the double trees comprise a better separator. It remains to choose $p_1$ and $p_2$ such that the claim about the smallest eigenvalues of the crossproduct holds, and to show that the resulting cut is bad.

Set $p_1$ to some arbitrary $p$, subject to the conditions presented below to insure that $p$ is sufficiently large. Then set $p_2 = \left\lceil p^{\left(\frac{1}{2}+2\epsilon\right)} \right\rceil$. Note that $p$ can be chosen sufficiently large such that

$$p > p^{\left(\frac{1}{2}+2\epsilon\right)} + 1 > p_2.$$

This implies that $p > n^{\frac{1}{2}}$, where $n = p_1 p_2$. Note that this allows one to show easily that $n^\epsilon < p^{2\epsilon} < p_2$ (i.e., the argument requires the path graph to have at least that many eigenvalues, and thus be at least that long). Also note that even for fairly small $p$, $p_2 < 2p^{\left(\frac{1}{2}+2\epsilon\right)}$, which implies that

$$n < 2p^{\left(\frac{3}{2}+2\epsilon\right)}. \tag{2.3}$$

Now consider the ratio of the size of the cut produced by cutting the double-tree edges to the size of the cut produced by a purely spectral method under the assumption that the $n^\epsilon$ smallest eigenvalues are the same as for the path graph. As in previous proofs, this ratio is at least as large as the ratio between the number of edges cut. Thus, for sufficiently large $p$, the ratio is at least

$$\frac{p}{\left\lceil p^{\left(\frac{1}{2}+2\epsilon\right)} \right\rceil} > p^{\left(\frac{1}{2}-2\epsilon\right)} - 1.$$

Using the fact that $p$ is also big enough that $p > n^{\frac{1}{2}}$,

$$p^{\left(\frac{1}{2}-2\epsilon\right)} - 1 > n^{\frac{1}{2}\left(\frac{1}{2}-2\epsilon\right)} - 1 = n^{\left(\frac{1}{4}-\epsilon\right)} - 1.$$

All that is left to prove is the assumption about the smallest eigenvalues. If $\alpha = \frac{1}{2} - 2\epsilon$, then $\alpha > 0$ and inequality 2.3 above can be written as

$$n < 2p^{(2-\alpha)}. \tag{2.4}$$

Recall that the eigenvalues of a path graph on $k$ vertices are $4\sin^2(\frac{\pi i}{2k})$ for $0 \leq i < k$, and that $\lambda_2$ for a double tree on $p$ vertices is greater than or equal to $\frac{1}{p}$. It remains to show that for $p$ sufficiently large,

$$4\sin^2\left(\frac{\pi n^\epsilon}{2\left\lceil p^{\left(\frac{1}{2}+2\epsilon\right)}\right\rceil}\right) < \frac{1}{p}.$$

Reorganizing, simplifying, noting that $\sin(\theta) < \theta$ for $0 < \theta \leq \frac{\pi}{2}$, and applying inequality 2.4 above, it is sufficient to show that there is a sufficiently large $p$ such that

$$\frac{\pi n^\epsilon}{2\left\lceil p^{\left(\frac{1}{2}+2\epsilon\right)}\right\rceil} < \frac{\pi\left(2p^{(2-\alpha)}\right)^\epsilon}{2p^{\left(\frac{1}{2}+2\epsilon\right)}} = \frac{\pi 2^\epsilon p^{-\alpha\epsilon}}{2p^{\frac{1}{2}}} < \frac{1}{2p^{\frac{1}{2}}}, \text{ or } \pi 2^\epsilon < p^{\alpha\epsilon}.$$

Clearly this inequality holds for sufficiently large $p$.
□

### 2.6.2 A Final Note About Tree-Cross-Path Graphs

While the tree-cross-path graph appears to be very specialized, the following argument suggests that it has some relation to practice: Section 3 noted that the Second Interlacing Theorem implies that adding an edge to a graph $G$ gives a new graph $G'$ with $\lambda_2'$ greater than or equal to $\lambda_2$ for $G$. Therefore the preceding results holds if each tree in the double trees is replaced with a graph that has a complete binary tree as a spanning tree (the edge between the two graphs connects the vertices at the roots of the spanning trees, and connections between copies of the "double graphs" in the crossproduct are between corresponding vertices in the spanning trees). One could therefore construct a three-dimensional finite element mesh from our example that represents the channel tunnel (the Chunnel) between England and France; the chunnel is a pair of long tubes with a small connection between the center.

## 2.7 Spectral Methods: Summary and Contributions

In this chapter I have provided some initial analysis of the quality of the separators produced by spectral methods based on the eigenvectors of the Laplacian matrix. In particular, I have shown that spectral bisection can do very poorly even on bounded-degree planar graphs, and that there are graphs for which using threshold cuts based on the second smallest eigenvector can do as poorly as theoretically possible. I have also shown that generalizing the algorithm and allowing the use of a constant number of smallest eigenvectors does not improve the quality of the separators for these families of graphs.

I have also introduced some facts about the structure of eigenvectors of Laplacian and symmetric matrices, particularly those with automorphisms of order 2. I have also shown how those facts can be used, both in demonstrating the structure of the second smallest eigenvector, and in proving upper and lower bounds on $\lambda_2$ for complete binary trees and double trees.

A discussion of future work based on these results is included in Chapter 4.

# Chapter 3

# Planar DAG Reachability

In this chapter, I introduce the Poincaré index formula, and show how it can be used in the design and analysis of a reduction procedure for planar DAGs. I also show how this reduction procedure can be used to implement a many-source reachability algorithm for planar DAGs, which in turn can be used in a multi-source reachability algorithm for planar digraphs that has a faster running time than previous published versions.

## 3.1 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 methods for solving this problem are breadth-first search (BFS) and depth-first search (DFS). They only require time 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. For sparse graphs the situation is better, though still not optimal with respect to work: Ullman and Yannakakis give a probabilistic parallel algorithm that works in $O(\sqrt{n})$ time using $n$ processors [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 [Kao93], Kao and Shannon [KS89] [KS93], and Kao and Klein [KK90] showed that the reachability problem and many related problems could be solved in polylogarithmic time using only a linear number of processors for planar digraphs. The planar reachability problem for multiple start vertices is specifically addressed in [KK90]. That paper presents a series of reductions between related problems; each reduction introduces more logarithmic factors to the running time. In the end, it takes $O(\log^5 n)$ time to solve this problem.

In this thesis I give a general method for reducing planar directed acyclic graphs (DAGs) to a constant size. I show that after $O(\log n)$ rounds of reduction an $n$-node directed planar DAG is reduced to a constant size. There have been several reduction rules proposed for undirected planar graphs [Phi89, Gaz91] but this is the first set for a class of directed planar graphs. Once the rules

for reduction have been presented, it is a relatively simple matter to "overlay" rules necessary to compute multiple-source reachability.

The results in this thesis 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 class of directed planar graphs are important for at least two reasons. First, the class includes several important subclasses including tree and series parallel graphs. Second, the flow graphs for many structured programming languages without function calls are planar. My goal is to develop the basic algorithmic foundation for a class of planar graphs in order to support a theory of planar flow graphs.

This thesis presents the details of the reduction technique for the planar DAG case. The algorithm for planar DAGs is interesting in its own right. First, it alone is sufficient to improve the computation of many-source reachability by a factor of $\log^2 n$ time by simply using the strong connectivity of Kao [Kao93] (our algorithm for general planar digraphs should remove 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. Finally, the algorithm itself contains many interesting ideas. (In Chapter 4 I do briefly discuss changes needed to extend the reduction procedure to general planar digraphs.)

Throughout the thesis I assume that the graph $G = (V, A)$ is a directed embedded planar graph. If an embedding is not given it is possible to construct one in $O(\log n)$ time using $n$ processors using the work of Gazit [Gaz91] and Ramachandran and Reif [RR89]. I assume that the embedding is given in some nice combinatorial way such as the cyclic ordering of the arcs radiating out of each vertex.

The following six sections cover the reachability algorithm and related results. The next section (Section 3.2) gives the main definitions necessary to define and analyze the directed graph reduction algorithm. Section 3.3 gives the reduction algorithm for special case of of a planar DAG. The theorems in Sections 3.4 and 3.5 show that the reduction algorithm for planar DAGs works in a logarithmic number of reduction steps: Section 3.4 shows that at any step of the reduction process, a constant fraction of the edges are candidates for removal or contraction; Section 3.5 shows that a constant fraction of these candidates can be operated on without affecting a set of invariants required of the graph's structure. Section 3.6 explains how the reduction procedure can be applied to the many-sources reachability problem and calculates the running time. Finally, in Section 3.7 I discuss the contribution of this work. Future work is discussed in Chapter 4.

## 3.2 Preliminaries

### 3.2.1 Planar Directed Graphs

I 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 $V \times V$. Arc $a = (u, v)$ is directed from $u$ to $v$; $u$ is the **tail** and $v$ is the **head** of the arc. 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; I represent this number as degree($v$). The **indegree** of a vertex $v$ is the number of arcs that have $v$

as their head; the **outdegree** of $v$ is the number of arcs with $v$ as their tail.

For any directed graph $G$, I define an undirected graph $G'$ on the same set of vertices in the following way: for each arc $(u, v)$ in $G$, include an edge $(u, v)$ in $G'$. I refer to $G'$ as the **underlying graph** of $G$. In this chapter I distinguish between edges and arcs: edges are undirected and lie in the underlying graph, while arcs are directed. When I refer to arcs in $G$ as edges, I am actually referring to the associated edges in $G'$. (The notation $E$ represents the set of edges of an undirected graph.)

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 the case $v_0 = v_k$ is allowed) and for all $0 < i \leq k$ the arc $(v_{i-1}, v_i)$ is 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)**.

For a directed path $p$ that is not a cycle, the **rank** of a vertex $v$ on $p$ is the number of vertices on $p$ that precede $v$.

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. There may be many different ways to draw a digraph in the plane; any particular way can be specified by giving the cyclic ordering (either clockwise or counterclockwise) of the arcs incident to each vertex. Such a specification is called a **planar embedding** of the digraph.

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 adjacent to that face. The **size** of a face is the number of arcs encountered in a traversal of the face's boundary (note that a single arc could be counted more than once in the size of some face). The set of faces is denoted by $F$. (The definitions of these terms are essentially the same for an undirected embedded planar graph.)

In an embedded planar digraph I define **parallel arcs** as two arcs that form a face of size 2. **Parallel edges** in an embedded planar graph are defined in the same way. This relation can be extended by making it transitive; in that case I say that a set of arcs or edges is **mutually parallel**.

There is an important formula developed by Euler (not surprisingly referred to as **Euler's formula**) that relates the numbers of edges, vertices, and faces in embedded, connected planar graphs:

$$|V| - |E| + |F| = 2. \tag{3.1}$$

If the graph is also simple (i.e., it has no loops and no parallel edges) and has 3 or more vertices, then each face has at least three edges in its boundary, and it is easy to prove the following inequality:

$$|E| \leq 3 \cdot |V| - 6. \tag{3.2}$$

Proofs that these formulas hold can be found in basic graph theory textbooks (e.g., [BM76]). The formula corresponding to (3.1) (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 (3.2) (with $|A|$ substituted for $|E|$) holds for an embedded planar digraph $G$ with a connected underlying graph if $G$ has no loops or parallel edges. Note that this implies that it holds for embedded planar DAGs with connected underlying graphs if the DAGs contain no parallel arcs.

### 3.2.2 The Poincaré Index Formula

Let $G(V, A)$ be a connected embedded planar digraph with faces $F$. 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 of the arcs (i.e., "out" to "in" or vice versa) as the arcs incident to that vertex are considered in cyclic order. 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 its 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 3.1. The alternation number of a face can be defined in a similar way. Here the number

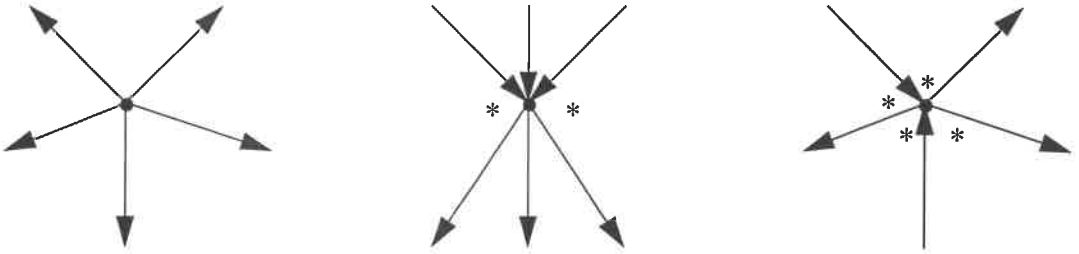



Figure 3.1: Vertex Types

of alternations is the number of times the arcs on the boundary change direction as that boundary is traversed. 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 3.2 below. I denote the alternation number of vertex $v$ by $\alpha(v)$, and the

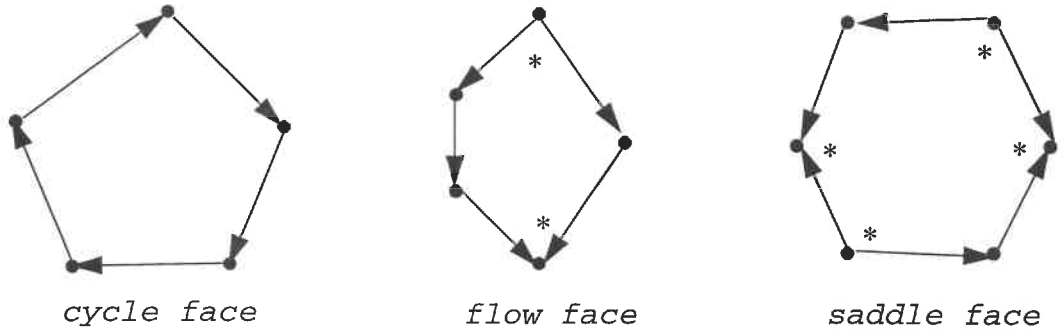



Figure 3.2: Face Types

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

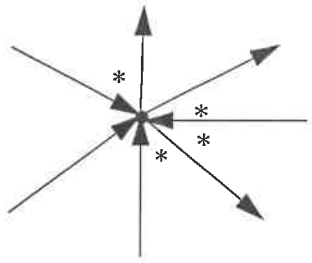
A concept related to alternation number is **index**. The index of a vertex $v$ (denoted $index(v)$) is defined as $\text{index}(v) = \alpha(v)/2 - 1$. The corresponding definition holds for the index of a face. Once again the usage will be clear from context.

My approach depends on combinatorial arguments based on the following simple but fundamental theorem that I refer to as the **Poincaré index formula**. (The name is drawn from the Poincaré index formula from combinatorial topology that relates the number of critical points of a vector field inside a closed curve to the winding number of the curve; intuitively, the arcs of the graph are like vectors and the vertices like critical points.) I show that it follows directly from Euler's formula.

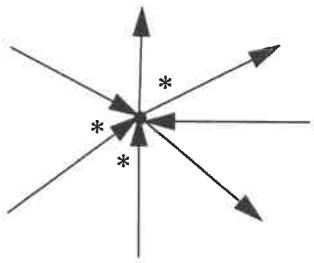
**Theorem 3.2.1** *For every embedded connected planar digraph,*

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

**Proof:** Consider the situation at any vertex as one cycles 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 3.3). If the



vertex alternations at a vertex



face alternations at a vertex

Figure 3.3: Alternations

number of alternations is summed over all vertices, 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 arcs in the graph:

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

Dividing by two and applying Euler's formula gives

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

which gives

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

which with some rearrangement and an application of the definition of index gives

$$\sum_{v \in V} \left( \frac{\alpha(v)}{2} - 1 \right) + \sum_{f \in F} \left( \frac{\alpha(f)}{2} - 1 \right) = \sum_{v \in V} \text{index}(v) + \sum_{f \in F} \text{index}(f) = -2.$$

□

This formula implies a great deal about the structure of a planar digraph embedding. For example,

- Sinks, sources, and cycle faces each contribute $-1$. These are the only structures that make negative contributions to the sums in the formula; since the total must be $-2$, it is clear that every embedded planar digraph must have at least two of them. 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; they contribute 0 to the sums in the formula. There can be an arbitrary number of such structures. It is easy to see that a flow face has two alternations on its boundary, one of which looks like a source with respect to the boundary, the other of which looks like a sink. Thus, at most one source and at most one sink can lie on the boundary of a flow face.
- Saddle vertices and saddle faces have positive indices that depend on their alternation numbers. Since the formula total 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.

I use the formula below to develop invariants and to help us count (for example, I use it to count particular types of arcs).

### 3.2.3 Models of Parallel Computation

The reduction algorithm is specified for the **Parallel Random-Access Machine (PRAM)** model of computation using concurrent reads and concurrent writes (i.e., the CRCW PRAM). I 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 ends up in that location).

## 3.3 Graph Reduction

In this section I introduce a collection of reduction rules and an associated data structure for planar DAGs. The reduction rules allow a graph to be converted into a smaller graph in order to recursively solve a problem. Once the problem is solved for the reduced graph, the graph is expanded out in reverse order to generate a solution for the original graph. In Sections 3.4 and 3.5 I show that at each stage the reduction process removes a constant fraction of the arcs; thus, the rules could

be implemented as an $O(\log |A|)$-step reduction procedure for planar DAGs. Since I require that the inputs have no parallel arcs, inequality (3.2) in Section 3.2.1 thus implies that the reduction procedure takes $O(\log n)$ steps (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 involve algorithm-specific actions performed as each rule is applied; I specify such actions in the algorithm descriptions in a later section.

I assume that the input is a connected, embedded planar DAG $G$ that has no parallel arcs (and hence no parallel edges). $G$ is preprocessed such that it has the following properties (these properties remain true throughout the algorithm):

1. $G$ has only flow faces. This is 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 3.4). It is straightforward to show that the number of arcs and hence the number of vertices increases by at most a constant factor.

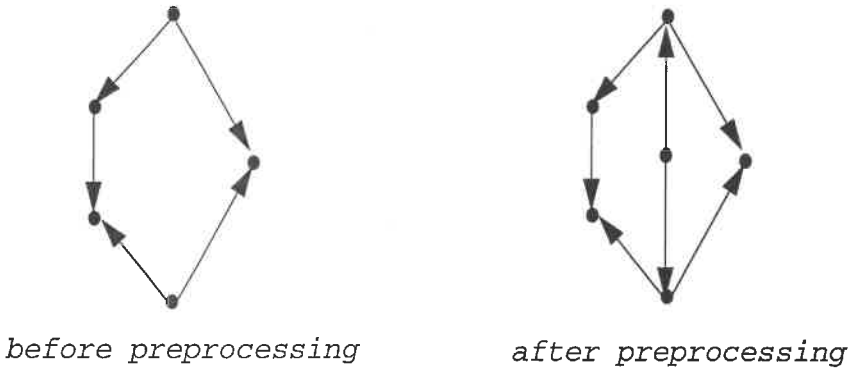



Figure 3.4: Preprocessing Saddle Faces

2. No vertex has both indegree and outdegree 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.

It is 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 of the graph.

### 3.3.1 Terminology

In order to simplify the presentation of the reduction rules, I 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 (I refer to any arc that is both on the left and the right path of a particular face 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 path 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. I mark top arcs with "T" and bottom arcs with "B."

Applying the rules may modify the connectivity of the graph. Therefore I associate flow faces with a data structure that allows connectivity information to be maintained. For each vertex on a flow face that is neither a top or bottom vertex there is 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 the algorithm also keeps the highest and lowest vertex on the opposite side of the face that points to this vertex (initially the high point in is set to bottom and the low point in is set to top).

For both the left and right path of each flow face, the top arc serves as the **leader** of the path (if the top arc is internal it serves as leader for both sides); each arc knows the two faces common to it and the leaders on those faces. Leader is defined similarly for topological arcs: the leader is 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 is maintained on each topological arc.

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, cross-pointers can now be tested in constant time to see if they have become forward pointers: simply test if the head and tail are on the same side of the face. (The coordination actions I use take constant time in the CRCW model.)

Saddle vertices are referred to 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. I call such arcs **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 I say that $a$ is **locally unique-in**; a symmetric definition holds for **locally unique-out**. Note that "locally" implies 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, the algorithm needs to distinguish certain unique-in and locally unique-in arcs out of a source. Such an arc $a$ out of a source is **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

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

that the second condition prohibits pointers across adjacent faces out of the tail of the arc.

The operation of **arc contraction** is defined 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 arc was.

### 3.3.2 Reduction Rules

I now list the reduction rules:

**[TB Rule]** If an arc $a$ is marked both T and B then remove $a$. If $a$ is topological, it may have crosspointers incident to its internal vertices. A crosspointer into an internal vertex $v$ is adjusted by a process referred to as **pointer splicing**: the cross-pointer into $v$ is set to point to the vertex pointed to by the cross-pointer out of $v$ on the opposite side of $a$. The remaining pointers are unchanged. Information on the structure of the face must be updated (e.g., the left and right leaders must be updated), and any new or changed topological arcs must be updated. Pointer updating is shown in Figure 3.5 (the lighter arrows indicate pointers, the darker ones arcs). **[Degree-1 Rule]**

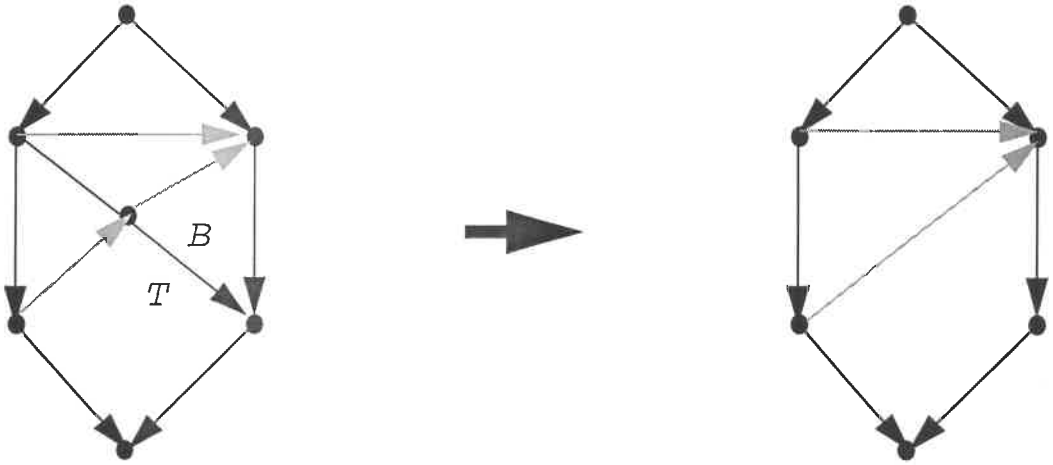



Figure 3.5: TB Rule Pointer Splicing

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 clean unique-in arc out of a source, contract $a$. The leaders on the affected faces are reset as 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 3.6, remove the source and sink and

their arcs as shown.

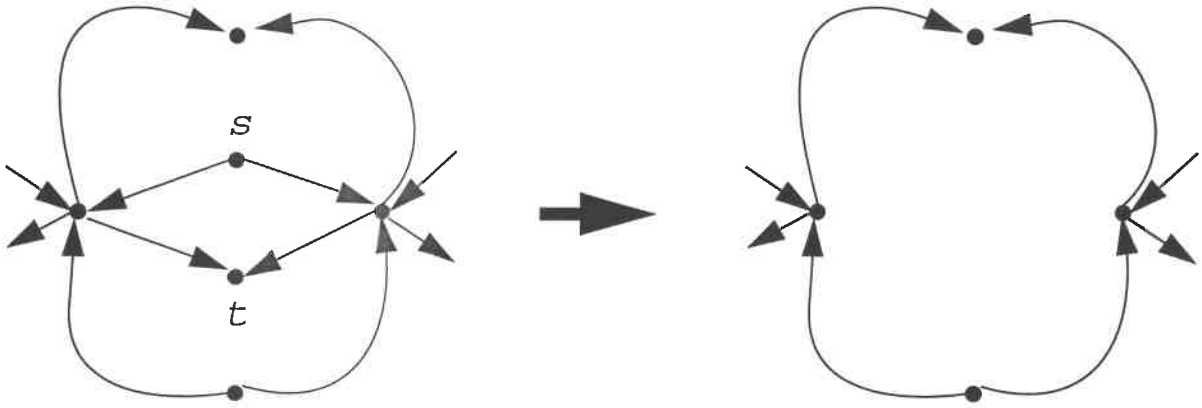



Figure 3.6: Adjacent Degree-2 Sources and Sinks Rule

**[Source-Sink-Source (s-t-s)/Sink-Source-Sink (t-s-t) Rule]** Let $s$ be a degree-2 or degree-3 source incident only to clean locally unique-in arcs. Further, at two of the saddle vertices $u_1$ and $u_2$ adjacent to $s$ let there be locally unique-out arcs (respectively $a_1$ and $a_2$) such that $a_1$ ($a_2$) is adjacent in the cyclic order at $u_1$ ($u_2$) to the arc incident to $s$. If $a_1$ and $a_2$ are also incident to distinct sinks $t_1$ and $t_2$, 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 3.7 – since all faces are flow faces, each sink is 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 (Figure 3.8).

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.

**[Consecutive Rule]** Let $s$ be a source incident to a clean locally unique-in arc $a$. At the head of $a$, if the next arcs in both the clockwise and counterclockwise directions are clean locally unique-out arcs into sinks, do the following: remove $a$, and combine the two sinks into a single sink (see Figure 3.9). A corresponding rule applies for a sink and adjacent sources.

**[Index-1 Saddle Rule]** If a source has a clean arc to a saddle vertex of index 1 and if the

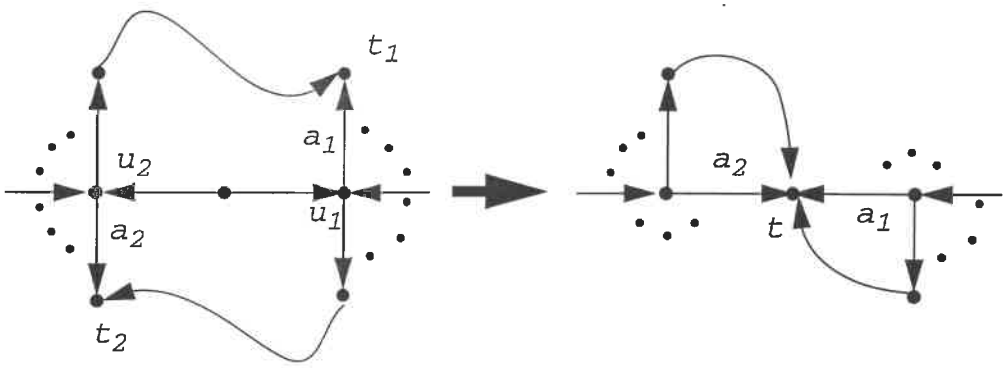



Figure 3.7: Sink-Source-Sink Rule (Degree-2 Source)

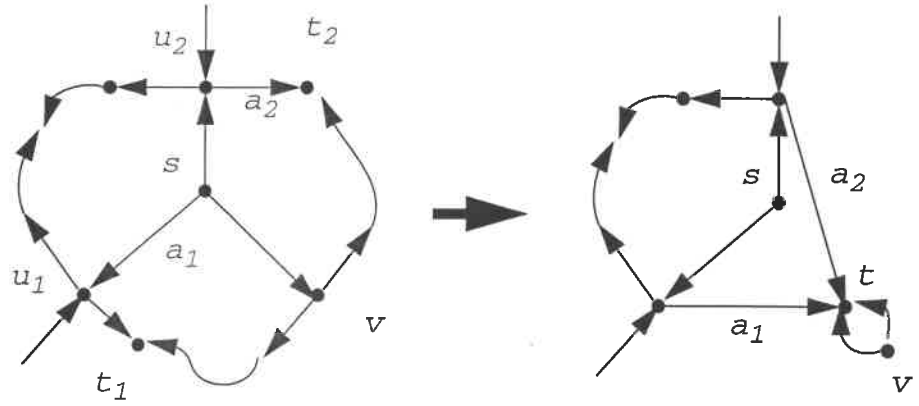



Figure 3.8: Sink-Source-Sink Rule (Degree-3 Source)

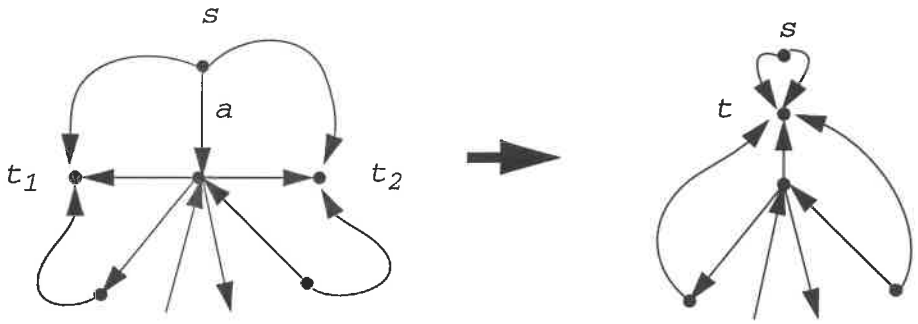



Figure 3.9: Consecutive Rule

only other arc into the saddle is a clean arc from another source, then contract one or both[2] of the in-arcs (see Figure 3.10). A corresponding rule holds if there are exactly two clean out-arcs to sinks (Fig. 3.11). As for the s-t-s and t-s-t Rules, a processor must be selected to represent that vertex if a large number of vertices are combined into a single vertex.

If a source (sink) of degree 2 or 3 has two clean arcs to (from) a single index-1 saddle, note that these two arcs divide the plane into two pieces. Split the graph into two pieces as follows:

- the arcs and vertices in the interior piece of the plane (i.e., the piece not including the exterior face), including the vertex that was an index-1 saddle in the graph prior to rule application; and
- if the source or sink has degree 2, the arcs and vertices in the exterior piece of the plane (including the vertex that was the saddle prior to rule application); if the source or sink has degree 3, the arcs and vertices in the exterior piece of the plane with the third source arc incident to the vertex that was the index-1 saddle. The resulting graph in either case corresponds to the result of deleting the arcs and vertices in the interior piece and then contracting the two arcs incident to the saddle.

Each of the two graphs has strictly fewer arcs than the graph prior to splitting (see Figure 3.12 below).

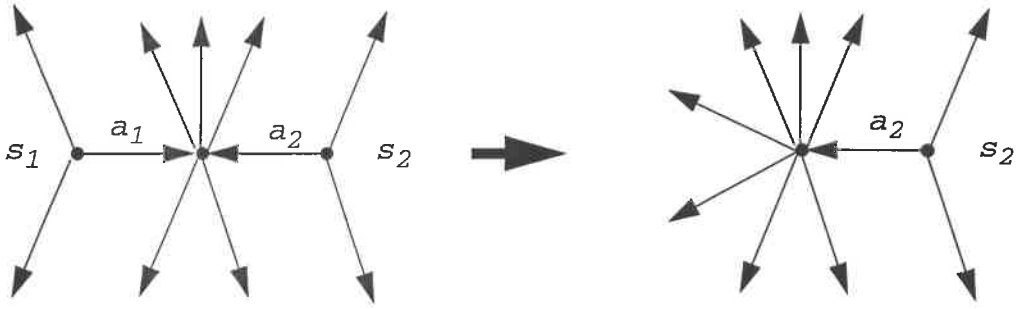



Figure 3.10: Index-1 Saddle Rule Applied to Sources – only arc $a_1$ contracted

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.

### 3.3.3 Cleaning Up the Graph

Many of the rules above operate only on clean arcs. Arcs in the configurations corresponding to particular rules are not necessarily clean, however. Therefore I introduce a parallel algorithm for

[2]Whether one or both are contracted is determined by the conflict resolution procedure as discussed in Section 3.5.1.

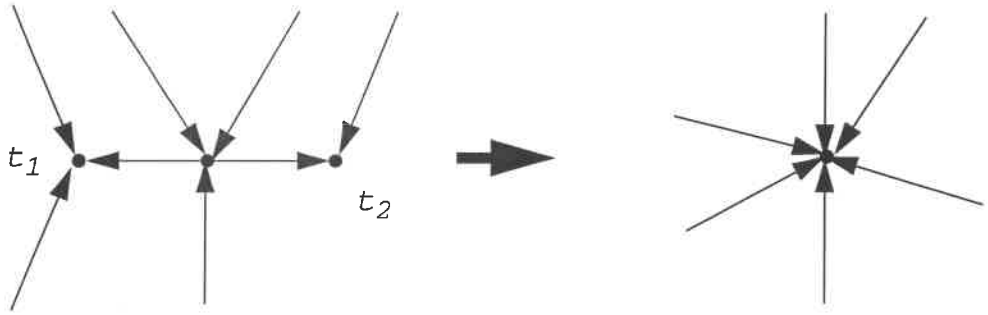



Figure 3.11: Index-1 Saddle Rule Applied to Sinks – both arcs contracted

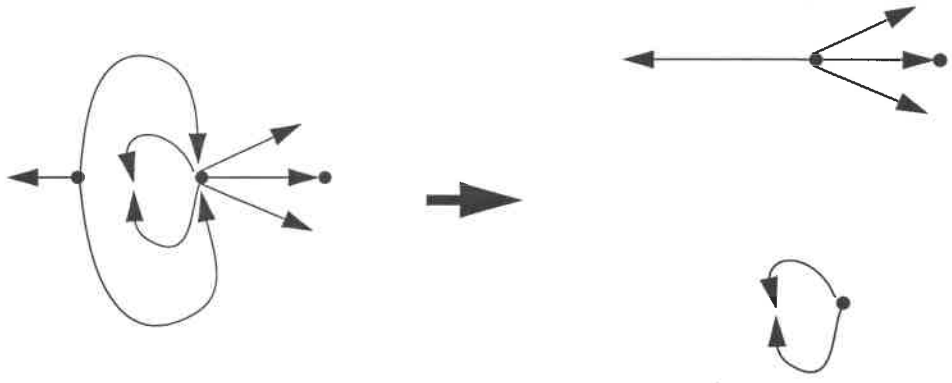


Figure 3.12: Index-1 Saddle Rule – Single Degree-3 Source Case

cleaning up arcs out of sources(into sinks) that runs in constant time in the CRCW model. The cleanup algorithm is run as a subroutine in the reduction algorithm. Because the cleanup algorithm can change the structure of the graph, it may be required to preserve some invariant specific to the problem being solved (e.g., in the case of many-source reachability the invariant is that the vertices in the current version of the graph that are reachable from one of the original sources are either marked as reachable or have a path consisting of pointers and arcs from some vertex with an active mark). Applying cleanup with respect to the invariant adds computation to the cleanup algorithm; in general, one should try to choose an invariant in such a way that it does not increase the asymptotic running time of the basic cleanup algorithm.

Not all sources and sinks are cleaned up. To insure that cleanup does not take too long (i.e., cleanup activities other than application-specific processing should take constant time), only sources (respectively sinks) of degree less than or equal to a constant $d$ (to be specified later) that are incident only to unique-in or locally unique-in (respectively unique-out or locally unique-out) arcs are cleaned. Note that such sources and sinks are not incident to parallel arcs. I explain the rationale for these restrictions in Section 3.4; intuitively, a planar DAG has few high-degree sources and sinks that are not incident to parallel arcs. No arcs are removed from such vertices because the arcs are not clean; however, since there are few such sources and sinks, this is not a problem.

### 3.3.4 The Cleanup Algorithm

Define the **frontier** of a source $s$ as the set of vertices at the heads of the arcs out of $s$. The frontier of a sink $t$ is the set of vertices at the tails of the arcs into $t$.

The cleanup algorithm consists of several steps. Application-specific processing can be added prior to or after any step; I defer further discussion to Section 3.6, where I discuss a particular application in detail.

At the first cleanup step, each source (respectively sink) determines if it has degree less than or equal to $d$, and if so, whether all incident arcs are either unique-in or locally unique-in (respectively unique-out or locally unique-out). Any source or sink not meeting these conditions drops out of the cleanup algorithm.

At the second cleanup step, for each arc out of each source still involved in the cleanup it is necessary to find the highest vertex on that arc (including its head) that can be reached from some other frontier vertex. Here "highest" means closest to the source, and "reached" means there exists a path of pointers from the frontier vertex to the high point, where each pointer has as its head and tail some vertex that lies between the source and the frontier (inclusive of frontier vertices). It is also necessary to compute which frontier vertex can reach this high point, and whether the pointer path proceeds in a clockwise or counterclockwise direction relative to the source. These things can be determined by following the high pointer chains out of each frontier vertex first in the clockwise direction and then in the counterclockwise direction. If more than one frontier vertex can reach the high point with respect to a single direction (e.g., clockwise around the source), the one at the greatest distance in terms of the cyclic ordering is chosen. The high point can be determined by comparing the location of the clockwise and counterclockwise high points relative to the rank ordering along the topological arc. If the same high point can be reached from both the clockwise and counterclockwise directions, the tie can be broken arbitrarily.

The case for sinks is symmetric. For each arc into each sink still involved in the cleanup it is necessary to find the lowest vertex on that arc (including its tail) that can reach some frontier vertex. Here "reach" means there exists a path of pointers from the low point to the frontier vertex, where each pointer has as its head and tail some vertex that lies between the frontier and the sink (inclusive of frontier vertices). Again, it is necessary to compute which frontier vertex can be reached from this low point, and whether the pointer path proceeds in a clockwise or counterclockwise direction relative to the sink. These things can be determined by following the chain of low pointers in reverse order, first in the clockwise direction and then in the counterclockwise direction. Note that if more than one frontier vertex can be reached from the low point with respect to a single direction (e.g., counterclockwise with respect to the sink), the one at the greatest distance in terms of the cyclic ordering is chosen. The low point can be determined by comparing the location of the clockwise and counterclockwise low points relative to the rank ordering along the topological arc. If the same low point can reach the frontier in both the clockwise and counterclockwise directions, the tie can be broken arbitrarily.

Note that for any cleaned source there must be at least one arc $a$ out that has no high point. Similarly, for any cleaned sink there must be at least one arc $b$ such that $b$ has no low point. To see this for sources, note that if an arc $a$ has a high point there is another frontier vertex $v$ that has a path to the head of $a$ consisting of the pointer path to the high point plus the segment of $a$ between the high point and the head. One can construct a directed graph consisting of the frontier vertices for this source and arcs representing the existence of a path from one frontier vertex to another. Since the original graph is a DAG, the constructed graph must be acyclic. But if every arc out of the source has a high point, then every frontier vertex in this graph has an arc in, which contradicts the fact that it is acyclic.

At the third step the graph is realigned as shown in Figure 3.13 below (the presentation here is in terms of sources; the actions for sinks are symmetric). First consider arcs that are reachable

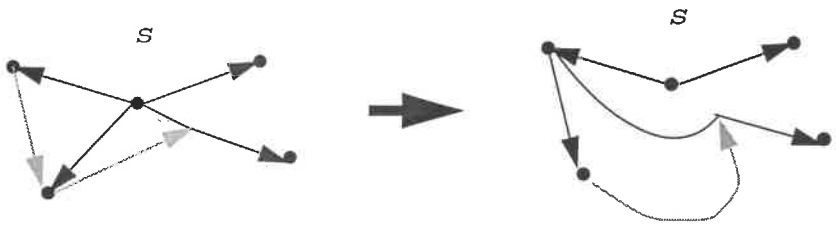



Figure 3.13: Cleanup: Realignment at a Source

from another frontier vertex: Each arc (if the high point is a frontier vertex) or arc segment (if the high point is internal to a topological arc) from the source to the high point is replaced by an arc (or, if internal, an arc segment) from the most distant frontier vertex. If a frontier vertex reaches multiple high points, the cyclic order of the new arcs at the frontier vertex is same as the cyclic order of the deleted arcs to those vertices at the source. All pointers to vertices at or below the high point are retained.

For arcs that are not reachable from another frontier vertex, if the arc is topological, replace the arc with an arc containing no internal vertices (i.e., all internal vertices are deleted).

Each source and sink that has been cleaned up is now marked "cleaned up". Each arc out of a cleaned source or into a cleaned sink is marked as "cleaned".

It is easy to verify the following claims: Every cleaned source has at least one arc out; every arc out of a cleaned source is clean. Likewise every cleaned sink has at least one arc in; every arc into a cleaned sink is clean. All the resulting faces are flow faces. The number of arcs and vertices in the graph does not increase. The reachability for every vertex remaining in the graph is unchanged in the following sense: Let $u$ and $v$ be any two vertices that are not internal to topological arcs. If there is a path of arcs and crosspointers from $u$ to $v$ prior to cleanup, then there is such a path after cleanup.

In order to avoid any conflicts between arcs common to both a source and a sink these steps can be run twice, once for sources and once for sinks.

These steps could be implemented in a number of ways. Because the degree of any cleaned source or sink is bounded by a constant and because the leader of each topological arc can keep track of bookkeeping information, the cleanup algorithm can be programmed to execute in a constant number of steps in the CRCW model provided that any application-specific processing added executes in constant time.

### 3.3.5 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 the algorithm invariants.
2. Main Reduction Loop: While there are arcs left in the graph, repeat the following sequence of steps:
   - Clean up the current graph, performing any application-specific actions where needed.
   - Apply the reduction rules in the order they are listed in Section 3.3.2. 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 the algorithm has stored, in order, all changes made during the reduction process).

This process takes constant time given that the conflict resolution steps noted take constant time using randomization in the CRCW model; the deterministic algorithm takes time $O(\log^* n)$ for each reduction step. The proof that the graph is reduced by applying this process $O(\log n)$ times (thus giving an $O(\log n)$ time randomized algorithm or an $O(\log n \ \log^* n)$ deterministic algorithm) is presented in the rest of this chapter.

## 3.4 Operability Lemmas

In the next two sections I prove that the reduction procedure given above works in $O(\log n \log^* n)$ time using $O(n)$ processors, provided that the application-specific processing takes at most constant time per reduction phase. (I use the convention that $n = |V|$ throughout the rest of this section.) The main result of this section is that at each pass through the main reduction loop a constant fraction of the arcs are candidates for removal (I refer to such arcs as **operable**). I start with two preliminary lemmas.

**Lemma 3.4.1 [Flow Face Operability]** *An arc $a$ between two flow faces is operable if it is neither unique-in, locally unique-in, unique-out, nor locally unique-out.*

**Proof:** Such an arc $a$ must have an adjacent out-arc in the cyclic ordering at its tail vertex, which must be the top vertex of one of the flow faces. Therefore $a$ is a T arc. A symmetrical argument shows that $a$ is also a B arc. Thus, $a$ is operable by the TB rule. □

**Lemma 3.4.2 [Unique-In and Unique-Out Arc Count]** *In a connected, embedded planar DAG consistent with the algorithm invariants the number of unique-in and unique-out arcs not incident to degree-1 vertices is less than or equal to $2/3$ the number of arcs in the graph.*

**Proof:** Note that in an embedded DAG the unique-in arcs (respectively unique-out arcs) form a forest. For the purposes of this proof, the term **unique-in(unique-out) tree** refers to a maximal subgraph of such a DAG that consists of a tree induced by unique-in(unique-out) arcs in the DAG. The unique-in(unique-out) tree may contain vertices that have degree 1 in the original DAG; if these vertices and their incident arcs are deleted from the unique-in(unique-out) tree, the **trimmed unique-in(unique-out) tree** results. I start by counting the number of arcs to which each trimmed unique-in tree is incident in the current graph $G$ that either 1) are neither unique-in nor unique-out, or 2) are into degree-1 vertices, and proving that this number is greater than the number of arcs in the tree (the proof for unique-out trees is symmetric).

I first claim that every leaf $v$ of a trimmed unique-in tree must have at least two arcs out in $G$, each of which either is a unique-in arc to a degree-1 vertex or is neither unique-in nor unique-out. To see that there must be two or more arcs out, note that if $v$ were of degree 1 in $G$, that would contradict the fact that the tree is trimmed; if $v$ were of degree 2, the second arc would have to be an arc out, and $v$ would be a degree-2 flow vertex, contradicting the conditions of the lemma. I further claim that these arcs out of $v$ either are unique-in arcs to degree-1 vertices or are neither unique-in nor unique-out. Since there are two out-arcs, they cannot be unique-out; if they are unique-in they must be into degree-1 vertices or else this contradicts the assumption that $v$ is a leaf of a trimmed unique-in tree, which by definition is maximal. Therefore the claim must hold.

Next I pair each arc in the trimmed tree with a distinct arc $a$ in $G$ out of some tree vertex $v$ such that $a$ either is into a vertex of degree 1 or is neither unique-in nor unique-out. Note that each tree arc must be either into an internal node of the tree or into a leaf node. I pair each arc into a leaf $v$ with one of the arcs out of $v$ in $G$; this leaves one additional arc out of each leaf. To handle internal nodes, I introduce the following terminology: if an internal node has exactly one tree arc out, it is a *path node*; otherwise it is a *branch node*. (I do not count the root as an internal node, though it makes only minor technical differences in the statement of the following.) Each path node in a unique-in tree must have at least one arc out in $G$ that either is incident to a degree-1 vertex or is neither unique-in and nor unique-out; I pair one such arc with the unique tree arc into the path

node. The only arcs left to pair up are those into branch nodes, which I associate with distinct arcs from the set of arcs out of the leaves as follows: The number of leaves in a tree is easily shown to be greater than the number of branch nodes. Therefore, since there is exactly one unique-in arc into any branch node, there are fewer arcs into branch nodes than there are unpaired arcs left at the leaves. Thus all tree arcs are paired as claimed. The arcs associated with each trimmed unique-in tree arc are clearly distinct, and, since I only count arcs out of trimmed unique-in trees, no arc is counted for more than one such tree. Therefore there is at least one distinct arc of one of the two claimed types for every unique-in arc in the graph that is not incident to a vertex of degree 1. This completes the argument.

By a symmetric argument, there is either a distinct unique-out arc out of a node of degree 1 or a distinct neither-unique-in-nor-unique-out arc for every unique-out arc in the graph that is not incident to a vertex of degree 1. To finish the proof, observe that each neither-unique-in-nor-unique-out arc out of a unique-in tree could also be an arc into a unique-out tree; thus, in the worst case each of these arcs is counted twice. In that case the number of these arcs is at least $1/3$ the number of arcs in the graph, from which the lemma follows. □

I now state the main lemma of this section:

**Lemma 3.4.3 [Operability Lemma]** *In any connected, embedded planar DAG that has been cleaned up and that is consistent with the algorithm invariants, a constant fraction of the arcs are operable.*

**Proof:** The lemma follows immediately from Lemmas 3.4.4 and 3.4.8 below, which prove the result for the cases in which the number of sources and sinks is less than $n/14$ and greater than or equal to $n/14$ respectively. □

Before proving these two lemmas, I briefly sketch their proofs. Lemma 3.4.4 deals with the case in which the number of sources and sinks is less than a specified fraction of the number of vertices in the graph. Its proof first shows that there are many arcs that either are unique-in or unique-out and incident to degree-1 sources and sinks, or are neither unique-in nor unique-out. This follows from the Unique-In, Unique-Out Arc Count Lemma (Lemma 3.4.2 above). This is not quite enough to prove Lemma 3.4.4, however; it is necessary to show that most of the non-unique-in, 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. By the Flow Face Operability Lemma (Lemma 3.4.1 above), this is sufficient to show that a constant fraction of the arcs in the graph are operable by the TB Rule.

Lemma 3.4.8 covers the case in which the number of sources and sinks is at least a specified fraction of the vertices in the graph. I prove it using a counting argument. First I show that a high degree source or sink $v$ (i.e., a source or sink with degree greater than the constant $d$ introduced in Section 3.3.3) either is incident to a TB arc or is uncommon in the sense that the total number of such sources and sinks is less than a constant fraction of the total number of sources and sinks in the graph. Next I show that at least a constant fraction of the sources and sinks with degree $\leq d$ are incident to an operable arc. This is clearly true for such sources and sinks that are either degree-1 or incident to a TB or arc. Any other such sources or sinks are processed in the cleanup phase. I show that at least a constant fraction of the cleaned sources and sinks are incident to an operable arc by a counting argument based on the Poincaré index formula. I then show that, excluding parallel arcs, a constant fraction of the arcs are operable, and since all parallel arcs are both T and B (and

hence operable), the lemma follows.

I now proceed with complete proofs:

**Lemma 3.4.4** *In any connected, embedded planar DAG consistent with the algorithm invariants and in which the number of sources and sinks is less than $n/14$, $1/6$ of the arcs are operable.*

**Proof:** To prove this lemma I show that in graphs meeting the stated conditions there are many arcs that either are incident to degree-1 sources or sinks, or are operable by the TB Rule.

Let $k$ be the number of sources and sinks in the graph. To make the proof easier to read, I use the following notation to refer to specific sets of arcs:

- $A_1$ is the set of arcs incident to degree-1 vertices.
- $A_2$ is the set of arcs not in $A_1$ ($A_2 = A \setminus A_1$).
- $A_3$ is the set of arcs in $A_2$ that are neither unique-in nor unique-out. Lemma 3.4.2 can be stated as follows:
$$|A_3| + |A_1| \geq \frac{|A|}{3}.$$
- $A_4$ is the set of operable arcs in $A_3$.
- $A_5$ is the set of inoperable arcs in $A_3$ ($A_5 = A_3 \setminus A_4$).
- $A_{op}$ is the set of all operable arcs.

Before proceeding, it is useful to recall that graphs meeting the algorithm invariants have no degree-2 flow vertices (such vertices become parts of topological arcs), so every vertex other than a source or sink has degree 3 or more. Thus the number of arcs in the graph is at least $3(n-k)/2 + k/2 = 3n/2 - k$. I give a lower bound on $|A_1| + |A_4|$; since all arcs in $A_1$ are operable, this is a lower bound on the number of operable arcs.

To get a lower bound on the size of $A_4$, I first note that the graph only has flow faces, so every arc in $A_3$ lies between two flow faces. Thus by Lemma 3.4.1 an arc in $A_3$ is operable if it is not locally unique-in or unique-out. Recall that a locally unique-in or unique-out arc must be incident to a saddle vertex.

I now apply the Poincaré index formula. The graph has no cycle or saddle faces, so this involves only the indices of sources, sinks, and saddle vertices. Since sources and sinks each contribute $-1$ to the sum and saddles each contribute a positive amount, the total number of saddles is less than or equal to $k-2$. The formula implies that for each source or sink beyond the first two there are two alternations on some saddle vertex; there are also two additional alternations per saddle vertex. Thus, the graph can have at most $4k-8$ alternations at saddles vertices. I associate each alternation with a single arc in the following way: A vertex alternation is associated with a pair of arcs; associate the alternation with the second arc of the alternation with respect to the cyclic ordering of arcs around the saddle vertex (I refer to this arc as the one "following the alternation"; the first arc of the alternation "precedes the alternation"). Note that each locally unique-in or unique-out arc must have an alternation associated with it: such arcs are by definition immediately preceded and followed by alternations. Since each inoperable arc in $A_3$ is locally unique-in or unique-out, each has an alternation associated with it. Each alternation is associated

with exactly one arc, so there are at most $4k - 8$ inoperable arcs in $A_3$. This gives us an upper bound on the size of $A_5$; since $A_5$ and $A_4$ partition $A_3$,

$$|A_4| = |A_3| - |A_5| \geq |A_3| - 4k + 8 > |A_3| - 4k.$$

I now use the fact above with Lemma 3.4.2:

$$A_{op} \geq |A_1| + |A_4| > |A_1| + |A_3| - 4k \geq \frac{|A|}{3} - 4k.$$

The lemma follows when substitutions are made using the assumption that $k < n/14$ and the fact that $|A| \geq 3n/2 - k$:

$$A_{op} > \frac{|A|}{3} - 4k > \frac{n}{2} - \frac{13k}{3} > \frac{n}{6}.$$

□

To prove Lemma 3.4.8 and thus complete the proof of Lemma 3.4.3, I first need to introduce some terminology and preliminary lemmas. I assume that the graph has been cleaned up.

For analysis purposes I associate a value of $i$ with each saddle vertex, where $i$ is equal to the index of that saddle vertex. This value is distributed equally among the alternations at the saddle; each alternation gets $i/2(i+1)$. The alternations assign their values to sources or sinks in the following way:

- Value is assigned only to cleaned sources with only locally unique-in arcs out, or to cleaned sinks with only locally unique-out arcs in. I refer to such sources and sinks as **eligible**.
- Value from a particular saddle vertex is assigned only to eligible sources (or eligible sinks) that are the tails (respectively heads) of arcs incident to that saddle (i.e., only to eligible sources and sinks at distance 1 from that saddle).
- If only one source or sink can be assigned value from a particular saddle, that source or sink receives that saddle's full value. If value from a saddle can be assigned to more than one source or sink, it is done so in the following way: for each eligible source or sink at distance 1 from this saddle, count the number of alternations between it and the next such eligible source or sink in both the clockwise and counterclockwise directions around the saddle. The source or sink is assigned the value for half that number of alternations.

Clearly each saddle vertex assigns a total value of either 0 or its index to sources and sinks. Note that the minimum value that an eligible source or sink can be assigned per locally unique-in or locally unique-out arc is $1/4$.

I refer to the total value assigned to a source or sink as the **value** of that source or sink. Under certain conditions presented in Lemma 3.4.6, particular sources or sinks transfer some or all of their value to other sinks or sources. This transfer is done in such a way that the total value summed over all sources and sinks remains constant.

I call a source or a sink with a value of $9/8$ or greater **uncommon**; other sources and sinks are **common**. In the arguments below, I associate a distinct operable arc with each common source and with each common sink. Since each such arc can be associated with at most one common source and one common sink, this proves that the number of operable arcs is proportional to the number of common sources and sinks.

**Lemma 3.4.5** *In an embedded planar DAG with a total of $k$ sources and sinks, more than $k/9$ of the sources and sinks are common.*

**Proof:** It follows from the remarks above and the Poincaré index formula that the total value that can be assigned by all alternations at all saddle vertices is bounded above by $k-2$. Each uncommon source or sink gets value greater than or equal to $9/8$. If $8/9$ of the sources and sinks were uncommon, their total value would be greater than or equal to $(8k/9)\cdot(9/8) = k$, which is greater than the total value available for assignment. □

**Lemma 3.4.6** *In an embedded planar DAG meeting the algorithm invariants, every source incident only to clean locally unique-in arcs is either uncommon or at the tail of an operable arc. Similarly, every sink incident only to clean locally unique-out arcs is either uncommon or at the head of an operable arc.*

**Proof:** The sources and sinks meeting the conditions of the lemma are the eligible sources and sinks as described above. I argue on the basis of the degree of the eligible source or sink. Note first that if an eligible source or sink is of degree 1, the incident arc is operable by the Degree-1 Rule. Further, if an eligible source or sink is of degree 5 or greater, it is uncommon (the minimum value that an eligible source or sink can get from each adjacent saddle vertex is $1/4$). If an eligible source or sink has degree 3 or 4, then either one of the incident arcs meets the conditions for removal by the Consecutive Rule or it is uncommon (an eligible source or sink gets the value of at least 3/2 alternation (each with value of at least $3/8$) from any saddle vertex where the Consecutive Rule does not apply). Thus it is only necessary to prove that the result holds for eligible sources and sinks of degree 2 to complete the proof. I prove the result for the case of sources; the proof for sinks is symmetric. To simplify the arguments below, I refer to an eligible sink $t$ as **adjacent** to an eligible source $s$ at a saddle vertex $u$ if $t$ is incident to arc $a_t$, $s$ is incident to arc $a_s$, and $a_t$ and $a_s$ are adjacent in the cyclic order at $u$.

For eligible sources of degree 2 where each arc out is incident to a different saddle, the following cases apply:

- There are no adjacent eligible sinks at either saddle vertex. In this case the source gets the value of at least 4 alternations. If either saddle has index greater than 1 then the source is uncommon (recall that the value of an alternation at a saddle of index $i$ is $i/2(i+1)$). If both saddles are index 1 and the source has value 1, then each saddle must have an arc in from another eligible source. In this last case the Index-1 Saddle Rule holds and the source has an operable arc out.

- There is a single adjacent eligible sink (i.e., an eligible sink is at the end of exactly one edge out of one saddle). In this case the source gets the value of at least three and one-half alternations. There are three subcases: First, if both saddles have index of 2 or greater the source is uncommon. Second, if the saddle with no adjacent eligible sinks has index 1, then either there is another eligible source with an edge into that saddle (in which case the Index-1 Saddle Rule holds and the source under consideration has an operable arc out) or the source gets the value of all alternations at that saddle (in which case the source is uncommon). Third, the saddle without the adjacent eligible sink has index greater than 1 and the saddle with the adjacent eligible sink has index 1. Again, at the index-1 saddle either there is a second eligible source with edge into the saddle (in which case the Index-1 Saddle Rule

applies at the source under consideration) or the source gets a value of $1/2$ from this saddle and is uncommon.

- There are at least two adjacent eligible sinks. In this case there are two possibilities. If two of the sinks are distinct then either the t-s-t Rule or the Consecutive Rule holds and the source has an operable arc out. Otherwise the source is in the situation shown in Figure 3.14 below. There are numerous subcases to consider. In the first two the source has an operable arc out:

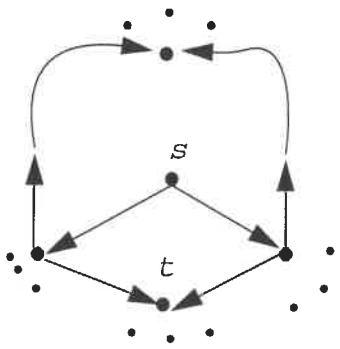



Figure 3.14: Degree-2 source with common adjacent eligible sink

  - The sink has degree 2. Then the Adjacent Degree-2 Sources and Sinks Rule applies and the source has an operable arc out.
  - The sink has degree 3 or greater and one saddle has index 1 and another eligible source with an arc in. Then the Index-1 Rule applies and the source has an operable arc out.

In the remaining subcases there are no rules that apply at the source and it must be shown uncommon (I refer to this as the problem source configuration:

  - the source has degree 2 and the arcs out are incident to different saddles;
  - a single eligible sink of degree 3 or greater is adjacent at each saddle; and
  - neither adjacent saddle has index 1 and is also adjacent to another eligible source).

Note that such a source has a value of at least 1 (it gets two alternations from saddles of index 1 and at least $3/2$ alternations at saddles of index 2 or greater). In some of these subcases value is transferred between sources and sinks as mentioned above. In all cases where value is transferred from sinks to sources, the sources must be in the problem source configuration. When a sink transfers value, it divides the value equally among all adjacent sources in such a configuration. Now consider the remaining subcases:

  - The sink is operable. In this case it transfers all its value. The sink has a value of at least degree$(t)/4$; since the number of problem configurations any sink can be involved in is less than or equal to its degree, each source in a problem configuration with this sink gets additional value of at least $1/4$, making it uncommon.

- The sink $t$ has degree 3 or greater and is not operable. In this case the sink transfers value equal to $1/4$ to the source, making the source uncommon. I show below that such a sink has sufficient value to transfer $1/4$ to all such sources to which it is adjacent and still remain uncommon.

For eligible sources of degree 2 where both arcs are incident to the same saddle, the following cases apply:

- If the saddle has index 1, then the Index-1 Rule holds and the arcs out are operable.
- If the saddle has index 2 and there are no adjacent eligible sinks with respect to the source, then the source gets at least 4 alternations and is uncommon.
- If the saddle has index 2 and there is at least one eligible adjacent sink, there are two subcases. The first is as follows: Let the two arcs out of the source be $a_1$ and $a_2$ respectively. The cyclic order around the saddle is divided into two segments, one between $a_1$ and $a_2$ and the other between $a_2$ and $a_1$ in clockwise order. Since the source is eligible, two of the alternations must fall in one segment and four in the other. Note that there can be an adjacent eligible sink in the segment with two alternations only if it is a degree-1 sink that is adjacent to both $a_1$ and $a_2$. In this case the Consecutive Rule applies. This is shown in Figure 3.15 below.

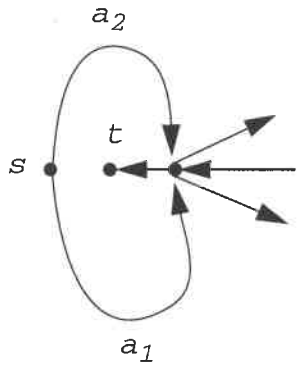



Figure 3.15: Adjacent eligible sink $t$ in two-alternation segment

- The only remaining case where the saddle has index 2 is when there are adjacent eligible sinks only on the segment with 4 alternations. Note that if there is not an eligible sink adjacent to both $a_1$ and $a_2$, then the source gets at least $7/2$ alternations (each with value of at least $1/3$) and is uncommon. Otherwise, there must be an arc to an eligible sink adjacent to each arc out of the source. Further, since the sink at the end of each such arc lies on the same face as the source, it must be a single sink (there is at most one sink on any flow face). This situation can be resolved by looking at the degree of the sink:

- If the sink is of degree 2, then the Adjacent Degree-2 Source and Sink Rule applies, and the arcs out of the source are operable.
- If the sink has degree 3 or greater, then this is a variant of the problem source configuration, and the sink transfers value of $1/4$ to the source, making the source uncommon. Once again, the proof that making such a transfer is reasonable is given at the end of the proof of the lemma.

- If the saddle has index 3 or greater, then either the Consecutive Rule applies, or else each arc out of the source gets $3/2$ alternations, which gives the source value at least $9/8$ and thus makes it uncommon.

To complete the argument for degree-2 sources, and thus complete the proof of the lemma, I must show that in the cases in which value is transferred, each inoperable vertex transferring value retains enough value to stay uncommon. I start by noting that all transfers are from sinks of degree 3 or higher to sources of degree 2, or (in the symmetrical argument for sinks) from sources of degree 3 or higher to sinks of degree 2. Thus there are no conflicting transfers. I prove that in the sink-to-source transfers, inoperable sinks retain enough value; the argument for source-to-sink transfers is symmetrical.

Note that each such sink has a value of at least $\text{degree}(t)/2$. The sink must receive value of at least $1/2$ for each arc out of a saddle (if the saddle has index 1 neither the Index-1 Rule nor the Consecutive Rule can apply to $t$, which is inoperable, so there is at most one adjacent eligible source at that saddle and $t$ gets value of at least $1/2$; if the saddle has index 2 or greater the Consecutive Rule cannot apply, so $t$ gets at least $3/2$ alternations (each with value of at least $1/3$), which supplies a value of $1/2$ or more for that arc).

Next consider a sink $t$ that transfers value to one or more sources. Observe that each arc $a$ into $t$ can be adjacent (in the cyclic order at the saddle at the $a$'s tail) to at most one arc out of some source (if not, the Consecutive Rule would apply at that sink, which is inoperable). Since each source that receives value from $t$ is incident to two arcs, each of which is adjacent to a distinct arc into $t$, the number of such sources can be at most $\lfloor \text{degree}(t)/2 \rfloor$. If $t$ transfers $1/4$ to each such source, $t$'s remaining value is at least

$$\frac{\text{degree}(t)}{2} - \frac{\text{degree}(t)}{2} \cdot \frac{1}{4} = \frac{3 \cdot \text{degree}(t)}{8},$$

which is greater than or equal to $9/8$ for sinks with degree 3 or greater. Thus the sinks that transfer value remain uncommon.

□

Lemma 3.4.6 deals with sources and sinks that have been cleaned up. However, the cleanup algorithm is not applied to all sources and sinks. Lemma 3.4.7 shows that there are not too many sources and sinks that have not been cleaned and that are not adjacent to an operable arc. This, along with the fact that there are not too many uncommon sources and sinks, allows a proof that the number of operable arcs is suitably large. The argument starts with some definitions and observations.

A **problem high-degree source** is a source of degree greater than the constant $d$ (introduced in Section 3.3.3) with all arcs out either unique-in or locally unique-in. A **problem high-degree**

**sink** is a sink of degree greater than $d$ with all arcs in either unique-out or locally unique-out. Such vertices are problems in the sense that they may have no operable arcs and they cannot be cleaned up in constant time.

A **simplified underlying graph** of an embedded planar DAG $G = (V, A)$ is the embedded planar graph $G'' = (V'', E'')$ that results when each set of parallel edges in $G'$ (the underlying graph of $G$) is replaced by a single edge. $G''$ has the following properties:

- Any problem high-degree source or sink in $G$ has no parallel arcs, and hence has the same degree in $G''$ as it has in $G$.
- All faces in $G''$ have boundaries of length 3 or longer because there are no loops and because faces in $G'$ with boundaries of length 2 are formed by parallel edges. Thus Inequality (3.2) in Section 3.2.1 holds.
- The number of vertices is the same in $G''$ and in $G$.

Given these facts, it is easy to prove the following lemma:

**Lemma 3.4.7** *In any embedded planar DAG consistent with the algorithm invariants the number of problem high-degree sources and sinks is less than* $6n/d$.

**Proof:** Let $l$ be the number of vertices in $G''$ that have degree greater than $d$. Because Inequality (3.2) from Section 3.2.1 holds for $G''$,

$$\frac{d \cdot l}{2} < |E''| < 3n \rightarrow l < \frac{6}{d} \cdot n.$$

Since every problem high-degree source or sink in $G$ has degree greater than $d$ in $G''$, the number of such sources and sinks must also be less than $6n/d$.

□

Now set $d = 1512$, which by the argument in the preceding lemma implies that the number of problem high-degree sources and sinks in the graph is less than $n/252$.

**Lemma 3.4.8** *In any cleaned-up embedded planar DAG G consistent with the algorithm invariants in which the number of sources and sinks is greater than or equal to* $n/14$, *a constant fraction of the arcs are operable.*

**Proof:**

As in previous proofs, let $k$ be the number of sources and sinks.

I start with some preliminary observations: The arcs can be partitioned into two sets: those that have another parallel arc and those that do not. Since $G$ has no cycles, every arc parallel to some other arc is both T and B with respect to the face common to the two arcs, and is hence operable by the TB Rule. From the discussion above about the simplified underlying graph $G''$, the number of arcs in $G$ without parallels plus the number of sets of mutually parallel arcs is less than $3n$ (this number is $|E''| < 3n$).

The goal is to specify a set of operable arcs such that the size of this set is a constant fraction of the number of sources and sinks in the graph (i.e., a constant fraction of $k$). The operable arcs specified correspond to edges in $G''$: they are either arcs without parallels or single representatives of sets of parallel edges. Because $k$ is at least a constant fraction of $n$ by the conditions of the

lemma, and because the arcs in $G$ that do not correspond to unique edges in $G''$ are all parallel arcs (and thus operable), specifying this set of operable edges implies that a constant fraction of the arcs in $G$ are operable.

Specify the set of operable arcs in the following way: For each source(sink) at the tail(head) of at least one TB arc, put one such arc in the set; the arc is operable by one of the TB Rules. (To be consistent with the condition that only one arc corresponding to any edge in G" is chosen, if the arcs specified for a source and a sink both come from the same parallel set, then a single arc representing the parallel set is used for both the source and sink.) If the source or sink has degree 1, then the incident arc is operable by the Degree-1 Rule and is added to the operable set. The remaining sources all have only unique-in and locally unique-in arcs out; the remaining sinks only have unique-out and locally unique-out arcs in. Ignore problem high-degree sources and sinks for the moment, and assume that all edges out of sources and into sinks are clean. Thus, for any other source incident to a unique-in arc or sink incident to a unique-out arc it is possible to add such an arc to the set of operable arcs because of the Unique-In/Unique-Out Arc Contraction Rule. This leaves only sources with clean locally unique-in arcs out and sinks with clean locally unique-out arcs in; by Lemma 3.4.6 every such source or sink is either uncommon or at the tail or head respectively of an operable arc.

An operable arc has been specified for every source or sink that is not either a high degree problem or uncommon. The number of problem high-degree sources and sinks is less than $n/252$ by Lemma 3.4.7 and the choice of $d$; this is less than $k/18$ by the conditions of this lemma. The number of uncommon sources and sinks is less than $8k/9$ by Lemma 3.4.5. Thus the number of sources and sinks for which an operable arc has not been specified is less than $17k/18$, which means that an operable arc has been specified for more than $1/18$ of the sources and sinks.

In order to complete the proof, I must show that there are not too many duplicate arcs in the set. Since every operable arc specified is incident to a source or a sink, duplicates are included only when an arc in the set is specified for both a source and a sink. In that case arc is included at most twice; thus, the size of the set is at least a constant fraction of $k$. □

## 3.5 Conflict Resolution

In the previous section I showed that in any embedded connected planar DAG meeting certain invariants a constant proportion of the arcs are operable once the graph has been cleaned up. However, applying the reduction rules to these operable arcs leads to two types of conflicts that the algorithm must deal with: **intra-rule conflict**, and **inter-rule conflict**.

Intra-rule conflict arises when a single rule is applied in parallel to all arcs operable by that rule. Doing so leads to cases in which either invariants are not maintained or in which the rule applications cannot be completed in some specified amount of time (for DAG reduction, this is constant time). For example, removing two arcs, both of which are marked T and B, can result in a graph that does not meet the invariant that all faces are flow faces (see Figure 3.16 below). Another potential problem is that removing multiple arcs via the TB Rule could leave an arbitrary number of arcs that must be combined into a single topological arc (see Figure 3.17 below). Updating the information for all the internal components (e.g., determining leader information and rank ordering)

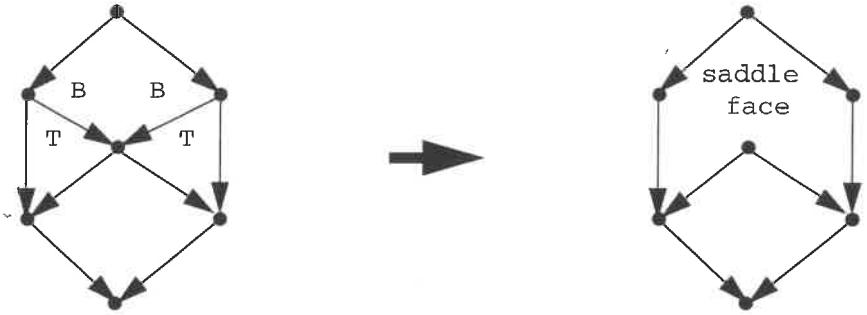



Figure 3.16: Example of intra-rule conflict for TB Rule

could thus take time $O(\log |A|)$. Likewise, it must be possible to combine faces in constant time

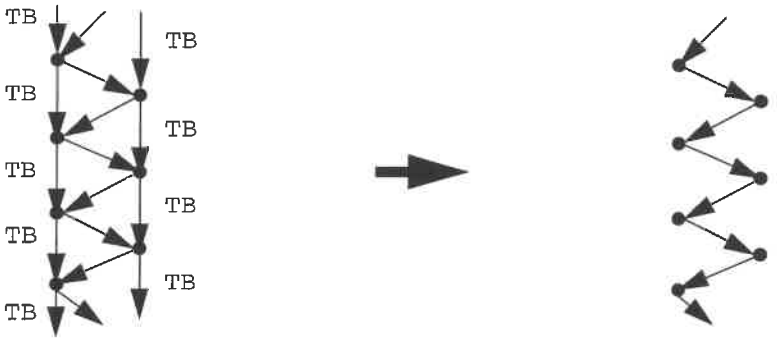



Figure 3.17: Creation of topological arc with arbitrarily many segments

(since rank orders are not kept on face boundaries, it is possible to combine an unbounded number of faces into a single face, but it must be shown that this does not require excessive time).

Inter-rule conflict arises when applications of a particular rule make arcs that were operable by another rule inoperable. Both types of conflict affect the counting argument that shows a constant proportion of the arcs are removed in each pass through the main loop.

Before discussing conflicts and conflict resolution in detail, recall the following observation made in Section 3.2.2, which is useful in a number of subsequent arguments: *A flow face has at most one source and one sink on its boundary.*

### 3.5.1 Resolution of Intra-Rule Conflict

The algorithm deals with conflicts between applications of a single rule by building a **conflict graph** that relates the conflicting arcs. The graph consists of a vertex for each arc operable by the rule in question, and an edge between each pair of these vertices where the removal of the corresponding

arcs causes a conflict. The edges can be undirected or directed, depending on whether the conflicts are symmetric or asymmetric. It is clear that choosing an independent set from the conflict graph gives a set of arcs that do not conflict with each other. I show how to find an independent set that includes at least a constant proportion of the vertices in the conflict graph, and thus a constant proportion of the arcs operable by a particular rule. In general, the conflict graphs have bounded degree, so finding a maximal independent set (MIS) in the conflict graph suffices.

The intra-rule conflict definitions for each rule follow. In all cases but one I state the maximum degree of the conflict graph. I argue that the "flow faces only" invariant is preserved. I also argue that any changes resulting from removing non-conflicting arcs can be processed in constant time in the CRCW model; in particular, I show that the algorithm never has to combine arbitrarily many arcs into a single topological arc, and that it never has to splice the arcs from arbitrarily many vertices into consecutive places in the cyclic order at some vertex. Also, I must show that where rules combine faces, the associated work can be done in constant time. The rule-by-rule conflict definitions are as follows:

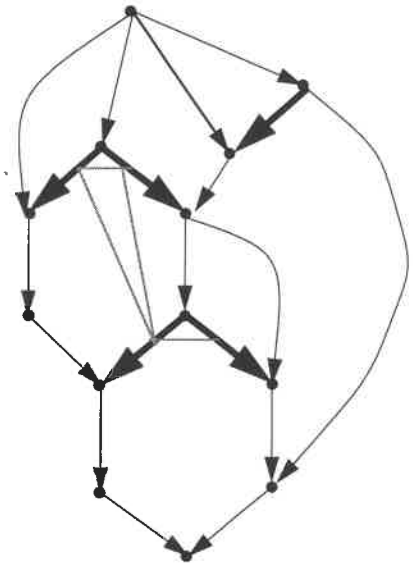
**[TB Rule]** For the TB Rule, the conflict resolution is broken into four stages. In each stage the algorithm determines conflicts for a particular type of TB arc, then removes non-conflicting arcs of that type. For purposes of the counting argument, I assume that the algorithm first determines all TB arcs, then applies the conflict resolution procedure. At the time that conflict resolution for a particular type of TB arc occurs, arcs of that type are specified. The reason for this is that as TB arcs are removed, the formation of topological arcs can change the characteristics of a particular arc. For example, an arc meeting the conditions for Type II TB arcs prior to arc removal could meet the conditions Type III TB arcs after the removal of a Type I TB arc. The four types are as follows:

- A **Type I** TB arc is not marked both T and B for any single face.
- A **Type II** TB arc is marked both T and B for exactly one face $f$, and the other T and B marks for $f$ are not common to a single arc.
- A **Type III** TB arc is marked both T and B for exactly one face $f$, and the other T and B marks for $f$ are common to a single arc.
- A **Type IV** TB arc is any TB arc that is marked both T and B for two faces.

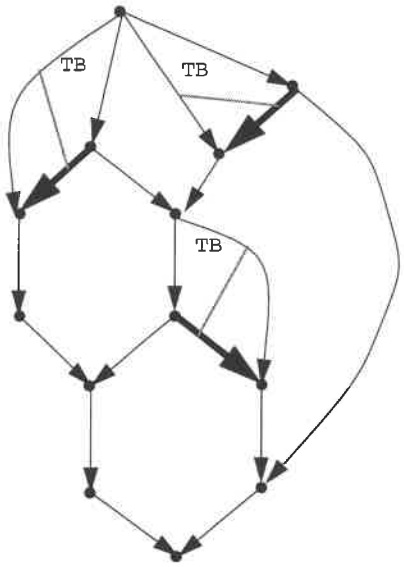
I describe the conflict resolution for each of these types in turn:

> **[Type I TB Arcs]** A Type I TB arc $a$ conflicts with any other Type I TB arc that is marked T or B with respect to a face for which $a$ is marked T or B. Each Type I arc conflicts with at most 6 other arcs (an arc can lie on two different faces and there are up to 3 arcs on each face with which it conflicts); these conflicts are symmetric (an example of a conflict graph is shown in Figure 3.18 below). A MIS is selected from the conflict graph and the associated arcs are selected for removal.
>
> Note that removal of these Type I arcs might either make certain Type II and Type IV TB arcs inoperable, or might make them into Type I arcs that are not removed during this arc-removal

phase. For this to happen, however, these arcs must be marked T, B, or both T and B on a face from which a Type I TB arc is removed. Thus vertices corresponding to these arcs can be added to the Type I conflict graph without increasing its maximum degree and without changing the fact that the MIS is maximal. (This extension of the conflict graph is not necessary in the actual algorithm; it is a counting mechanism used only in the proof. The fact that a Type II or Type III arc has become inoperable can be detected in a subsequent arc removal step.) An example of how Type II TB arcs are added to the conflict graph is shown in Figure 3.18.



*Conflict Graph for Type I TB Arcs: Heavy Arcs are Type I; Light Lines Indicate Conflicts*




*Results for Type I TB Arcs: Heavy Arcs are Type I MIS; Light Lines Indicate Type II Conflicts*

Figure 3.18: TB Rule Conflict Graphs

To see that this conflict resolution procedure preserves the "flow faces only" invariant, note that saddles result only if the removal of some set of arcs effectively changes the marks on some other removed arc so that it is no longer marked both T and B. To create such a conflict with a Type I TB arc $a$, it is necessary to remove an arc $a'$ that is common to a face $f$ with $a$, and that has the same mark as $a$ with respect to $f$. Since the algorithm only removes Type I TB arcs at this time, such an arc must be Type I, and the conflict procedure rules this out.

The TB Rule can never create cycle faces in a DAG, since it only removes arcs.

The following lemma shows that this procedure combines at most a constant number of arcs into a topological arc:

**Lemma 3.5.1** *At most 3 arcs are combined into a single topological arc as a result of*

*removing Type I TB arcs.*

**Proof:** Number the arcs that are incorporated into a new topological arc according to their order of occurrence on this new arc, with the arc closest to the tail being numbered 1. I refer to the $i^{\text{th}}$ arc as $a_i$. Suppose $v$ is the tail of some $a_i$, and that $v$ becomes internal to the new topological arc. Then every arc incident to $v$ other than $a_{i-1}$ and $a_i$ must be removed at the same time. Note that if two such incident arcs were adjacent in the cyclic order, the conflict resolution procedure for Type I TB arcs would prevent the removal of one of them. Therefore in the cyclic ordering at $v$ in the clockwise direction, there is at most one Type I TB arc between $a_{i-1}$ and $a_i$, and at most one between $a_i$ and $a_{i-1}$.

Now assume that removing Type I TB arcs causes four or more arcs to be combined into a single new topological arc (the following argument is illustrated in Figure 3.19 below). This implies that at least one arc incident to $v_4$, the tail of $a_4$, is removed. I refer to this arc as $b_4$. Without loss of generality, assume that $b_4$ lies between $a_3$ and $a_4$ in the clockwise cyclic order at $v_4$. $b_4$ and $a_3$ are common to a face $f_1$.

Another arc $b_3$ incident to $v_3$, the tail of $a_3$, must also be removed. If $b_3$ lies between $a_2$ and $a_3$ in the clockwise cyclic order at $v_3$, then it also is common to $f_1$, and conflict resolution prevents the simultaneous removal of $b_3$ and $b_4$. Therefore $b_3$ must lie between $a_3$ and $a_2$ in the clockwise cyclic order at $v_3$. $b_3$ and $a_2$ are common to a face $f_2$.

Finally, a Type I TB arc incident to $v_2$, the tail of $a_2$, must also be removed. However, if this arc lies between $a_2$ and $a_1$ in the clockwise cyclic order at $v_2$, it is common to face $f_2$ and conflicts with $b_3$; if it is lies between $a_1$ and $a_2$ in the clockwise cyclic order at $v_2$, it is common to face $f_1$ and conflicts with $b_4$ because there is no arc between $a_2$ and $a_3$ in the clockwise cyclic order at $v_3$. Thus the assumption leads to a contradiction.
□

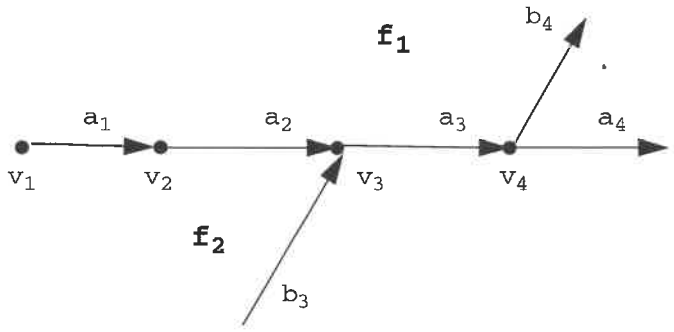



Figure 3.19: Example for Topological Arc Formation

To see that at most two crosspointers get spliced into one, first note that if more than two pointers are spliced together, then for any pointer other than the first or last, the arcs at both its head and tail must be removed. This requires the removal of two arcs from one face, which is prevented by the conflict resolution procedure.

Since the algorithm never removes more than one arc from any face, it is obvious that it never removes two arcs that are consecutive in the cyclic order at any vertex. Likewise, it never combines more than two faces into one.

**[Type II TB Arc]** The conflict resolution procedure for Type II TB arcs differs from the procedure for most other rules because it constructs two conflict graphs in sequence, and it differs from all other rules because the first conflict graph is a forest and may not have bounded degree. The vertices of the first conflict graph represent Type II TB arcs that remain operable after conflict resolution (including arc removal) for Type I TB arcs. (Recall that the algorithm considers arcs that were operable prior to the removal of any Type I TB arcs; any newly-created Type II TB arcs are considered inoperable, as are those made inoperable by Type I conflict resolution. Also note that the algorithm only considers arcs that are currently Type II; any operable Type II arcs that became Type III when Type I arcs were removed are considered later.) It is easy to see that any such arc meets two easily-tested conditions: it was operable prior to Type I removal, and it currently meets the conditions for Type II TB arcs. Each such Type II arc $a$ is on the boundary of a unique face $f$ for which $a$ is not both T and B. If the opposite side of $f$ is an operable Type II arc $b$, direct an arc in the conflict graph from the vertex representing $a$ to the vertex representing $b$. The result is a directed forest (see Figure 3.20). Removing Type II TB arcs does not affect the operability of Type III or Type IV TB arcs.

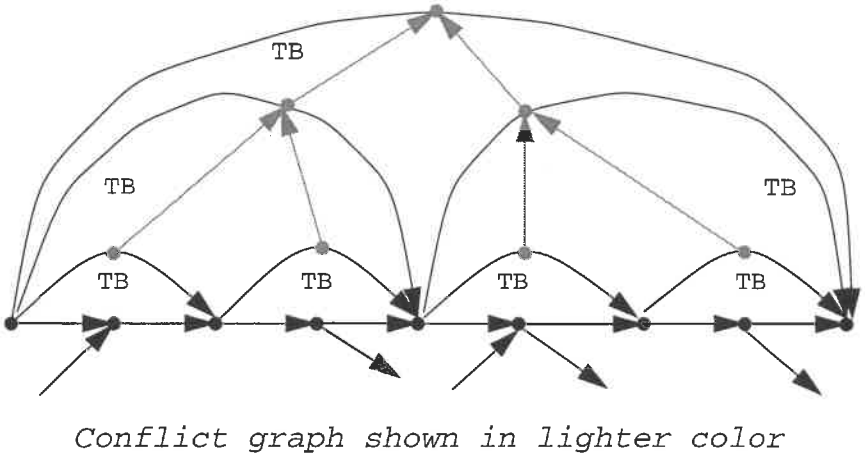



Figure 3.20: Conflict Graph for First Phase of Type II TB Conflict Resolution

Tree contraction could be used to find an independent set in the forest that contains at least

half the vertices. However, that could take time $O(\log |A|)$. Therefore the algorithm uses a simpler method that runs in constant time in the ARBITRARY CRCW model. To simplify the exposition, I first introduce some terminology: a **chain vertex** is any vertex in the conflict forest with indegree 1, provided that the arc in is not incident to a leaf. The first conflict resolution algorithm for Type II TB arcs can now be stated:

- Add all leaves in the forest to the set of arcs to be operated on.
- Form the subgraph induced by the chain vertices and replace the directed arcs by undirected arcs. This subgraph has maximum degree 2. Find a MIS in the subgraph and add the corresponding arcs to the set of arcs to be removed.

The argument that this yields at least $1/3$ of the operable Type II TB arcs is straightforward. I first note that since the subgraph induced by the chain vertices has maximum degree 2, the MIS has at least $1/3$ of these vertices. Next I count the vertices other than chain vertices. There are three types:

1. leaves
2. vertices with indegree 2 or greater. The proof to Lemma 3.4.2 noted that the number of such vertices must be less than the number of leaves.
3. vertices with indegree 1 for which the arc in is incident to a leaf. The number of such vertices is at most the number of leaves.

Thus the number of non-chain vertices is less than three times the number of leaves. Since all arcs corresponding to leaves are operated on, this is greater than $1/3$ of the remaining arcs, and the claim is proved.

The second phase of conflict resolution prevents the creation of topological arcs out of more than some constant number of arcs. The conflict rule is as follows: for each Type II TB arc $a$ selected in the first round of conflict resolution, consider the other T and B arcs on the face $f$ for which $a$ is marked both T and B. Each lies on the boundary of another face ($f_1$ and $f_2$ respectively; they need not be distinct). If the T arc is marked either T or B on $f_1$, and if the boundary of $f_1$ opposite from the T arc is a Type II TB arc marked both T and B on $f_1$ and chosen in the first round, put an edge between the vertices representing it and $a$ in the conflict graph. Likewise, if the B arc is marked either T or B on $f_2$, and if the boundary of $f_2$ opposite from the B arc is a Type II TB arc marked both T and B on $f_2$ and chosen in the first round, put an edge between the vertices representing it and $a$ in the conflict graph. These conflicts are symmetric, so the degree of any vertex in the conflict graph is at most 2, and a MIS from this graph corresponds to a set of arcs that is at least a constant fraction of the Type II TB arcs that were operable at the start of this conflict resolution stage.

To see that the two-step conflict resolution procedure preserves the "flow faces only" invariant, note that saddles result only if the removal of some set of arcs effectively changes the marks on some other removed arc $a$ so that $a$ is no longer marked both T and B. If $a$ is a Type II TB arc, this requires the removal of at least one arc marked either T or B on the face

for which $a$ is marked both T and B. Since only Type II arcs are removed at this stage, this is ruled out by the first conflict rule.

Removal of the arcs selected by this process causes at most three consecutive crosspointers to be spliced into one. If more than two pointers are spliced, then some pointer $p$ must lie between two arcs that are removed; these arcs must be common to a single face. The reduction rules insure that if two arcs are removed from a single face then neither is marked both T and B on that common face (if both are, they are not Type II TB arcs; if one is then the two arcs conflict in the first half of the conflict resolution step). Therefore, any pointer spliced to $p$ must cross a face to or from the side of a Type II TB arc that is marked both T and B. But the Type II TB arc is the only one removed from such a face, and the chain of splices cannot extend any further. Thus at most three pointers get spliced into one (one to a Type II arc, one from one Type II arc to a second, and one from the second Type II arc).

The same sort of argument shows that the maximum number of arcs consecutive in the cyclic order that are removed at some vertex is at most two. Any two such arcs are common to a face $f$. If the arcs have the same orientation (i.e., both are in-arcs), then by the argument above neither is both T and B on the common face. If they have opposite orientations, then neither can be both T and B on the common face. Therefore in either case they must be marked both T and B on the other faces; no other arcs are removed from those faces, so the next arcs in the cyclic order in either direction are not removed.

I still need to show that the number of arcs combined into a single topological arc is bounded by a constant. I have just shown that the algorithm never removes three consecutive arcs in the cyclic order at any vertex. I can apply this fact at any $v$ that becomes internal to a topological arc, so only cases in which one or two consecutive arcs are removed need be considered. If two consecutive arcs are removed, I can limit the possibilities to four as shown in Figure 3.21 below. The unlabeled arcs are components of the topological arc. In (a) both $a1$ and $a2$ could be Type II TB arcs and could both be removed without conflict. In (b) and (c), there is exactly one way in which both arcs could be Type II TB arcs; in those cases the arcs conflict by the first conflict resolution step. In (d) there is no way that the two labeled arcs could both be Type II TB arcs.

Now consider the possible configurations of arcs incident to a vertex $v$ that becomes internal to a topological arc. At least one arc incident to $v$ must be removed. The configuration of arcs at one side of $v$ (i.e., either clockwise in the cyclic order between the arc into $v$ and the arc out of $v$, or clockwise in the cyclic order between the arc out of $v$ and the arc into $v$) can be one of four things: no arcs, an arc in, an arc out, or two arcs as specified above. Note that if there is an in arc on each side of the topological arc at $v$ then these two arcs conflict by the second conflict resolution step; this is also the case if there is an arc out on each side. Therefore four possibilities (plus their mirror images) are left; they are shown in Figure 3.22 below.

Finally consider the configurations possible at two consecutive vertices $u$ and $v$ that become internal to a topological arc. Specifically, let there be an arc from $u$ to $v$ that becomes part of the topological arc, and let the arc out of $v$ be the last arc in the sequence that becomes

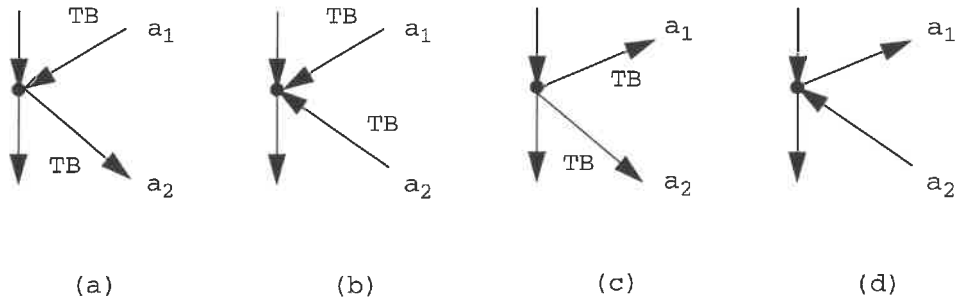



Figure 3.21: Consecutive Type II TB Arc Incidences at a Vertex

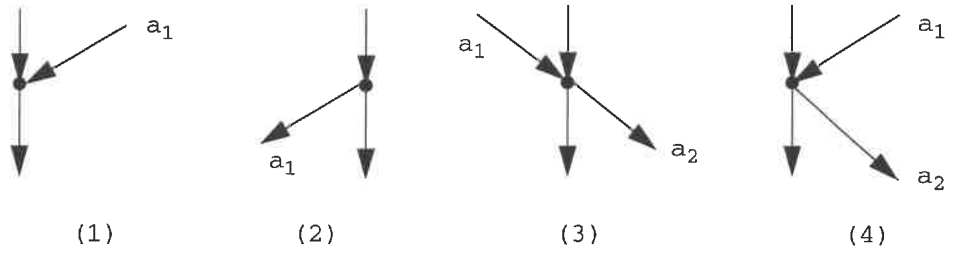



Figure 3.22: Nonconflicting Type II TB Arc incidences at vertices that could become internal

the new topological arc. First consider the case in which there is an in-arc $a_{in}$ incident to $v$. This in-arc conflicts with an out-arc on the opposite side of $u$, and an out-arc at $u$ on the same side of the topological arc as $a_{in}$ cannot be a Type II TB arc. If there is no out-arc at $u$, there must be an in-arc at $u$. However, such an arc conflicts with $a_{in}$ if it lies on the same side of the topological arc. Thus, if configurations (1), (3), or (4) shown in Figure 3.22, or their mirror images, occur at $v$, there is only one allowable configuration that can occur at $u$: configuration (1), with the arc on the opposite side of the topological arc from $a_{in}$. By the same argument, the only configuration that can occur at the vertex preceding $u$ on the topological arc is configuration (1), with the arc on the same side of the topological arc as $a_{in}$. But such an arc must conflict with $a_{in}$ by the first conflict resolution rule. Thus in this case at most three arcs can be combined into a single topological arc.

Now consider the case in which the configuration at $v$ is (2). If configuration (1), (3), or (4) occurs at $u$, the previous argument says that at most three arcs preceding the arc out of $v$ are combined into the topological arc, limiting the total number of arcs combined to four. If configuration (2) occurs at $u$, the arc out of $u$ must be on the opposite side of the topological arc from the arc out of $v$; if not, they would conflict by the first conflict resolution rule. This implies that configuration (2) cannot occur at the vertex preceding $u$; if one of the other configurations does occur at the preceding vertex, this is again the case discussed above, and at most three more arcs can be included in the new topological arc. Thus, at most five arcs can be combined into a single arc as a result of removing Type II TB arcs.

Type II TB arc removal is one case where an unbounded number of faces can be combined into one. Let $f$ be a face such that an arc marked both T and B with respect to $f$ is removed. The conflict resolution rules prevent $f$ from being combined with another such face. However, several faces such as $f$ can be combined with a face $f'$ such that no Type II TB arc marked both T and B with respect to $f'$ is removed. If this occurs, the processor for $f'$ becomes the processor for the new face. To create this new face, $f'$ needs to determine if the leaders have changed as a result of the removal of a Type II TB arc. Then the various faces to be combined with $f'$ need to set the processor of $f'$ as the new processor. The edges on the remaining boundaries of these faces can read the new processor number to complete the change. This can all be done in constant time in the CRCW model, so combining an arbitrary number of faces in this way is not a problem.

**[Type III TB Arc]** Recall that a Type III TB arc is marked both T and B for exactly one face $f$, and the other T and B marks for $f$ are common to a single arc $b$. Apply the conflict rules for Type III TB arcs to any such arc that is currently operable. Such an arc $a$ meets the following two conditions: $a$ was marked both T and B prior to the start of TB arc conflict resolution, and $a$ currently meets the conditions for Type III TB arcs.

The first conflict rule says that if $b$ is also an operable Type III TB arc, $a$ conflicts with $b$ and vice versa.

In addition, conflict rules are needed to prevent the formation of topological arcs from arbitrarily many arcs. To understand these rules, it is first useful to discuss the situations in which vertices can become internal as a result of Type III TB arc removal.

The first conflict rule assures that if Type III TB arc $a$ is removed, then an arc parallel to $a$ remains in the graph. This implies that the alternation number of any vertex in the graph does not decrease as a result of Type III TB arc removal. Thus only flow vertices can become internal to topological arcs.

Also, for each arc $b$ remaining after Type III TB removal, at most two Type III arcs parallel to $b$ have been removed. Thus if the outdegree (respectively indegree) of a flow vertex is four or more, that vertex cannot become internal as a result of the removal of Type III arcs.

The additional conflict rules thus apply to operable Type III TB arcs that are incident to flow vertices with indegree and outdegree less than or equal to 3. Let $a$ be such an arc and $v$ be such a flow vertex at the tail of $a$. Once again, $b$ denotes the TB arc common to the face for which $a$ is marked both T and B.

To determine these additional conflicts, consider the next arc in the cyclic order at $v$, where the direction of the order is specified as $b$ followed by $a$. If $c$ is out of $v$, then $a$ has no conflicts with respect to its tail. If $c$ is into $v$, the following cases apply:

- If $c$ is a Type III TB arc, the following conflicts can occur: $a$ and $c$ conflict if $c$ is operable; if the arc $d$ opposite $c$ on $c$'s TB side is an operable Type III TB arc, then $a$ also conflicts with $d$ (this last condition applies whether or not $c$ is operable).
- If $c$ is not a Type III TB arc, then consider the indegree of $v$. If the indegree is greater than one, $a$ has no more conflicts with respect to $v$. If $c$ is the only arc in, then consider the vertex $u$ at the tail of $c$ (if the indegree of $v$ is greater than one in this case, $v$ does not become internal). If $u$ is a flow vertex with indegree 3 or less and $c$ is the only arc out, then consider the arcs into $u$ (if $u$ does not meet these conditions, $u$ does not become an internal vertex). Let $d$ be the arc into $u$ on the face common with $a$. If $d$ is a Type III TB arc, the following conflicts can occur: $a$ and $d$ conflict if $d$ is operable; if the arc $e$ opposite $d$ on $d$'s TB side is an operable Type III TB arc, then $a$ also conflicts with $e$ (this last condition applies whether or not $d$ is operable). If $d$ is not a Type III TB arc, no conflicts occur.

These conflicts are illustrated in Figure 3.23 below. A symmetric set of conflicts occur for arcs out of the head of $a$.

As in previous cases the algorithm builds a conflict graph with vertices corresponding to the Type III TB arcs, and edges corresponding to conflicts. In this case conflicts may not be symmetric; however, for any operable Type III TB arc there are at most five arcs with which it can conflict and that can conflict with it. The degree of any vertex in the conflict graph is at most five. Thus it is possible to find a constant proportion of the vertices in the conflict graph by finding a MIS. As was the case for Type I arcs, removal of Type III arcs can affect the operability of other types of TB arcs. In particular, certain Type IV TB arcs can either become inoperable, or can become Type III TB arcs that are not removed during this phase of arc removal. However, all such Type IV arcs are common to a face for which a removed Type III arc is marked both T and B. As was the case for Type I conflict resolution, the conflict graph can be extended to include these arcs after a MIS has been selected: If $a$

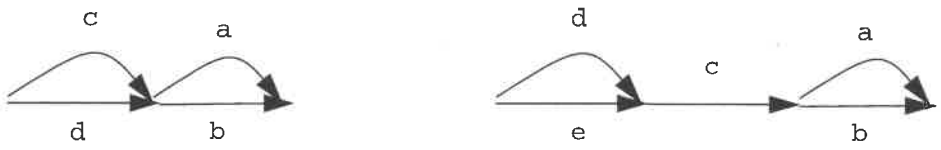



Figure 3.23: Type III TB Arc Conflicts

is represented by a vertex in the MIS in the conflict graph, and if $a$ is labeled both T and B on a face $f$, and the opposite side of $f$ is a Type IV TB arc $b$, add a corresponding vertex and edge to the conflict graph (again, this is done for counting purposes in the proof and need not be done in the actual algorithm). This does not increase the maximum degree of the conflict graph, and, since every added edge is incident to an element of the MIS, the MIS remains a MIS.

To see that this preserves the "flow faces only" invariant, note that saddles result only if the removal of some set of arcs effectively changes the marks on some other removed arc $a$ so that $a$ is no longer marked both T and B. This can only occur for Type III TB arcs if the algorithm removes two arcs both of which are marked both T and B on a common face. This is ruled out by the conflict rules.

Removal of the arcs selected by this process causes at most three consecutive crosspointers to be spliced into one. If more than two pointers are spliced, then some pointer $p$ must lie between two arcs that are removed; these arcs must be common to a single face. The conflict rules insure that if two arcs are removed from a single face then neither is marked both T and B on that common face (if either is, then by the definition of Type III TB arcs both are, and they conflict). Therefore, any pointer spliced to $p$ must cross a face to or from a the side of a Type III TB arc that is marked both T and B. But the Type III TB arc is the only one removed from such a face, and the chain of splices cannot extend any further. Thus at most three pointers get spliced into one (one to a Type III arc, one from one Type III arc to a second, and one from the second Type III arc).

It is easy to see that no more than two arcs consecutive in the cyclic order at any vertex are removed. The conflict rules insure that if a Type III arc $a$ is removed, then the other arc on the face for which $a$ is marked both T and B is not removed. If two consecutive Type III arcs are removed, then neither can be marked both T and B on the face they share. Further, the next arc in either direction in the cyclic order remains.

To see that no more than three arcs are combined into a single topological arc, first recall that if a vertex becomes internal, either its outdegree, its indegree, or both are reduced to one

(either the indegree or the outdegree could be one already). Because of the first conflict rule, if the outdegree (respectively indegree) is reduced, one can conclude that it was two or three prior to removal, and that the removed arcs were parallel to the remaining arc. It is easy to see that if a vertex $v$ has both indegree and outdegree of two or three, then the additional conflict rules insure that $v$ does not become internal (an example of such a vertex is included in Figure 3.24 below as the "excluded" case; the arcs labeled $a$ and $b$ conflict since the other two arcs must both be Type III TB arcs even if they are no longer operable).

This allows the number of cases considered to be reduced. There are four basic cases as shown in Figure 3.24 below (cases 1 and 3 each represent two mirror-symmetric configurations). Assume that $v$ is the last vertex that becomes internal in the newly formed topological arc.

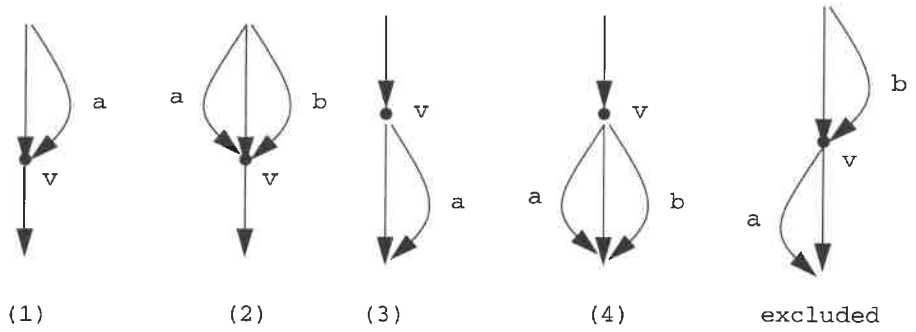



Figure 3.24: Cases for showing that Type III TB conflict resolution prevents the formation of long topological arcs

Let $a_{top}$ be the arc with $v$ as its head that is included in the topological arc, and let $u$ be the tail of $a_{top}$. If $u$ also becomes internal and the configuration at $v$ is either 3 or 4, then the configuration at $u$ has to be 1 or 2 respectively (operable arcs out of $u$ would be arcs into $v$; this is not consistent with the assumed configuration). In either case conflicts occur and not all the arcs incident to $u$ and $v$ are removed (note that for configurations 1 and 3, the arc parallel to the Type III TB arc must also be a Type III arc). If the configuration at $v$ is 1 or 2, then either configuration 3 or 4 occurs at $u$ (the arcs out of $u$ are the arcs into $v$); in this case its clear that at the next vertex back from $u$ on the desired topological arc there are conflicts as described above. In this case it is possible to combine three topological arcs into a single arc.

Removal of Type III TB arcs can lead to the combination of an unbounded number of faces in the same way as for Type II TB arc removal. The method of combination and the argument that it takes constant time are the same as in the Type II case above.

**[Type IV TB Arc]** Recall that a Type IV TB arc is marked both T and B for two faces. The Type IV arcs that are removable at this step meet the easily-tested conditions that they were marked both T and B prior to TB arc removal, and that they currently meet the conditions for Type IV TB arcs.

Each Type IV TB arc $a$ lies on the boundary of two faces. If $a$ is operable, it conflicts with any operable Type IV TB arc that forms the opposite side of either of these faces. Thus each such arc can conflict with at most two other arcs, and all conflicts are symmetric. The conflict graph has a maximum degree of 2.

It is easy to see that two arcs consecutive in the cyclic order are never removed: if the opposite sides of the faces bounded by a removed arc are not Type IV arcs, they are unaffected; if they are Type IV arcs they conflict with the removed arc and are not removed. Thus at most one arc is removed from any face. When a single Type IV TB arc between two faces is removed, the two faces are merged into a single flow face, so no saddle faces are formed. Also, at most two pointers can be spliced into one; if longer pointer chains were formed, two arcs would have to be removed from some face, which is not possible.

It is also clear that removal of a Type IV arc cannot create a topological arc. In general the conflict rules assure that both the top and bottom arcs on two faces stay incident to the head and tail of the removed arc. In the degenerate case in which the graph consists of two parallel arcs, the arc that remains is not combined with any other arc.

The conflict procedure for Type IV TB arcs prevents the removal of more than one such arc from any face. Therefore at most two faces are combined. This can easily be done in constant time.

This completes the conflict rules for various types of TB arcs. I now argue that this four-step procedure is sufficient to remove a constant fraction of the TB arcs: First note that the maximum degree of the conflict graph for Type I TB arcs is 6, so the conflict resolution procedure discussed in Section 3.5.1 above yields a MIS that is at least $1/7$ the size of the set that includes (1) arcs operable by this rule and (2) Type II and Type IV TB arcs that are not removed because of the removal of Type I arcs corresponding to vertices in the MIS. The arcs not in this set are Type II, Type III, and Type IV TB arcs that remain operable after Type I removal.

The conflict resolution procedure for Type II TB arcs removes a constant proportion of the remaining operable Type II arcs without affecting the operability of the remaining Type III and Type IV TB arcs.

The conflict resolution procedure for Type III TB arcs removes a number of arcs equal to a constant proportion of the remaining operable arcs that are either (1) Type III or (2) Type IV arcs that are not operated on because of the removal of Type III arcs. The argument is as for the Type I case.

Finally, the Type IV conflict resolution procedure allows the removal of a constant fraction of the remaining arcs. Thus a constant proportion of the arcs that were TB arcs prior to conflict resolution are removed.

**[Degree-1 Rule]** There are no conflicts between arcs operable by this rule, so the conflict graph has no edges. To see that no saddle or cycle faces are created, note that the removal of such an arc $a$ changes only the face of which $a$ is on the boundary. Furthermore, removing a degree-1 vertex and its arc causes the number of face alternations either to stay

the same or to decrease. Thus, given that all faces are flow faces, each starts with two alternations on the face. If the number of alternations goes down it must go to zero (there is always an even number of face alternations on a face), which would mean the face is a cycle. But that implies that the remaining boundary formed a cycle in the original graph, which contradicts the fact that the graph is a DAG.

Given that the graph being reduced contains no directed cycles and that there is at most one source and one sink on a flow face, it is easy to show that there are at most two arcs operable by this rule on any face and these arcs cannot be adjacent in the cyclic order at any vertex. There are no problems with processing time (i.e., with respect to pointer splicing or updating topological arc information) in this case.

The remaining rules affect only clean arcs. Therefore there is no need to worry about splicing pointers: A clean arc has no pointers through internal vertices, so its removal does not cause any splicing. If it is contracted, the head becomes top (respectively, the tail becomes bottom), and any incident pointers can be deleted (the conditions on clean arcs insure they do not become self-loops or backpointers). However, it is necessary to be careful that no more than a constant number of faces get combined into a single face as the result of the application of some rule.

**[Unique-In(Unique-Out) Arc Contraction Rule]** An arc $a$ operable by this rule conflicts with its two neighbors in the cyclic order at the source(sink). The conflicts are symmetric, so the degree of any vertex in the conflict graph is at most 2. Contraction of a unique-in(unique-out) arc does not create a cycle or saddle face: Only the two faces that have this arc on a boundary are affected. Since the next arcs in a traversal of these faces have the same orientation on the boundary as the contracted arc, the face remains a flow face, and the "all flow faces" invariant holds. The conflict rule insures that consecutive arcs in the cyclic order are not contracted, so changes in the cyclic order can be processed in constant time in the CRCW model. There are no problems with either combining too many arcs into a topological arc or combining too many faces together: no topological arcs can be formed and no faces can be combined by this rule.

(Note that there are no conflicts in which this rule applies to an arc at both a source and a sink; if a unique-out arc from a source is the unique arc into a sink, then these vertices plus the arc form a complete connected component.)

**[Adjacent Degree-2 Sources and Sinks Rule]** A conflict graph can be constructed as follows: Vertices in the conflict graph are the operable degree-2 sources. Each such source $s$ checks for each sink $t$ with which it is operable (there are at most 2) whether there is a second source $s'$ that is operable with $t$, and if so adds an edge to $s'$ in the conflict graph.

In addition, it may be the case that $s$ lies on a face $f$ that has a degree-2 sink $t$ as its bottom, and $s$ is not operable with $t$. However, $t$ may be operable with another source $s'$; if this is the case, then $s$ and $s'$ conflict. This conflict is defined in a symmetrical way with respect to $s'$: if $s'$ is operable with with a sink $t$, and if $t$ lies on a face $f$ such that the top of $f$ is a source $s$ that is operable, but not with $t$, then $s'$ and $s$ conflict. These conflicts are included to prevent

the combination of more than a constant number of faces or the creation of a topological arc from arbitrarily many arcs.

Since the conflicts are symmetric, the maximum degree of any vertex in the conflict graph is 2. If any source selected for removal during conflict resolution is operable with more than one sink, it chooses one of the sinks arbitrarily.

If only non-conflicting sources and their corresponding sinks are removed from a graph that contains only flow faces, the conflict rules insure that each removal affects only three faces: face $f_1$ for which the source is top and the sink is bottom; face $f_2$ for which the source is top and the sink is not on the boundary, and face $f_3$ for which the sink is bottom and the source is not on the boundary (these faces are easily identifiable in Figure 3.3 in Section 3.3.2). When the source and sink are removed, the remaining face consists of the paths from the top of $f_2$ to the two saddle vertices and the paths from the two saddles to the bottom of $f_3$. This forms a new flow face; the "all flow faces" invariant continues to hold. The conflict rules also insure that no more than these three faces are combined into a single face.

It is obvious that no more than two consecutive arcs in the cyclic order at any vertex are removed at once. Arcs are removed in pairs, so if more than two were removed simultaneously there would have to be at least two conflicting sources.

It is straightforward to show that at most three arcs are combined into a single topological arc. To see this, assume that four arcs could be combined into a topological arc consistent with the conflict rules, and let $u$, $v$, and $w$ be the vertices that become internal in the order from tail to head of the topological arc. Since $w$ is not already internal, some incident arcs must be removed. Assume that the removed arcs lie on a particular side of the topological arc. The conflict rules do not allow all arcs to be removed from either $v$ or $u$ on the same side of the topological arc. Therefore all arcs must be removed from $v$ and $u$ on the other side of the topological arc, and at least one arc must be removed at each of those vertices. But this would remove conflicting arcs, contradicting the assumption.

**[Source-Sink-Source (s-t-s)/Sink-Source-Sink (t-s-t) Rule]** To make the exposition simpler, I refer to the source involved in a potential application of the t-s-t rule as the "operable source", and the sink involved in a potential s-t-s Rule application as the "operable sink".

To prevent problems such as removing an arbitrary number of consecutive arcs in the cyclic order at some vertex or combining an arbitrary number of faces, the algorithm applies these two rules in sequence (assume the t-s-t Rule is applied first, though the order is not important). The specific procedure is as follows: first, mark all sources and sinks that are operable by these rules. Apply the t-s-t Rule (there are no conflicts between operable sources). Test whether the sinks marked operable remain operable, and apply the s-t-s Rule to those that do (again, there are no conflicts between operable sinks). I define conflicts between sources and sinks, though no conflict graph need be constructed - this is another case where conflicts are used for counting purposes only. To understand these conflicts, note that applications of the t-s-t Rule could leave a neighboring operable sink inoperable either because the number of neighboring sources drops to one, or because that sink ends up with too high a degree.

Thus, every operable sink conflicts with every operable source with which it is common to a face. Since operable sources have degree 3 or less, the number of sinks that can become inoperable by a single s-t-s Rule application is clearly bounded.

For a particular source at which the t-s-t Rule applies there may be more than one way to apply the rule. The source can arbitrarily pick one of the ways; this is not a conflict in the sense I use the term. The same holds for applying the s-t-s Rule at some sink.

Each merging of cyclic orders at a source (for the s-t-s Rule) or a sink (for the t-s-t Rule) occurs between two arcs that are not removed, so consecutive merges do not occur. Thus there is no problem if a particular source appears in multiple s-t-s Rule applications or if a sink appears in multiple t-s-t applications (note that a high-degree vertex may be created as mentioned in the discussion of this rule in Section 3.3.2).

To see that the remaining graph has only flow faces, note that the removal of the arc (or arcs) out of any source affects only the structure of the two faces it borders (the case for sinks is symmetric). These two faces are replaced by two new flow faces. The same observation makes it clear that no arbitrarily large set of faces is merged into one.

It is obvious that no topological arcs are formed.

**[Consecutive Rule]** Let $a$ be an arc that is a candidate for removal by this rule, and let the two faces of which $a$ lies on the boundary be $f_1$ and $f_2$. Then $a$ conflicts with any other arcs that lie on the boundaries of $f_1$ and $f_2$ that are operable by the Consecutive Rule. These conflicts are symmetric. The conflict graph in this case has maximum degree 6.

It is obvious that this rule prevents the removal of successive arcs in the cyclic order at a source, sink or saddle vertex, and that the cyclic order at combined sources or sinks can be updated in constant time.

It is easy to see that the application of this rule cannot produce saddle or cycle faces. Any single application of this rule affects only the two faces of which the removed arc lies on the boundary. These two flow faces are reconfigured into two new flow faces; the rest of the graph is unaffected. The same argument shows that a Consecutive Rule application never combines more than a constant number of faces into a new face. Also, no topological arcs are produced since all removed arcs are incident only to saddle vertices and sources or sinks.

**[Index-1 Saddle Rule]** An arc $a$ operable by this rule conflicts with at most its two neighbors in the cyclic order at the source(sink), provided that those neighbors are incident to different saddles. It also may conflict with two arcs adjacent in the cyclic order at the saddle if these arcs are incident to sinks(sources) at the saddle (i.e., when this rule is applied at a source there may be two sinks at the index-1 saddle that also are operable by this rule). The degree of any vertex in the conflict graph is at most 4 (the conflicts are symmetric).

This conflict rule insures that if an arc is contracted then none of the adjacent arcs in the cyclic order at either end are affected, so there is no problem with splicing cyclic orders. Applications of this rule that involve two sources(sinks) only contract arcs, so no topological

arcs are formed, and faces are not combined. In this case, the argument that no saddle or cycle faces are created is the same as for the Contraction Rule above.

Applications that involve a single source or sink adjacent to a saddle affect at most three faces. The separation of the graph does not create any topological arcs, and the affected faces all remain flow faces, though their boundaries are changed. The cyclic orders at the affected vertices can be modified in constant time.

With the exception of the first step of conflict resolution for Type II TB arcs, the degree of each vertex in the conflict graphs for each reduction rule is bounded by a constant. The conflict graphs are not necessarily planar, however (e.g., it is easy to construct graphs for which the Type I TB arc conflict graphs are not planar). All conflict graphs 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 fraction of the vertices. Therefore the algorithm can apply the techniques developed by Goldberg, Plotkin, and Shannon [GPS87] to resolve conflicts in $O(\log^* n)$ time.

If randomization is used, the running time can be reduced to constant time in the CRCW model. In particular, Luby's Monte Carlo Algorithm A (described in [Lub86]) can be used to find a MIS in constant time.

### 3.5.2 Conflict Resolution Between Rules

The second type of conflict is dealt with via the conflict resolution rules provided in the proof of the following lemma:

**Lemma 3.5.2** *Given the order of rule application specified in the algorithm description, a single application of any reduction rule reduces the number of arcs operable by subsequent rules (excluding the arcs removed by this rule application) by at most a constant number.*

**Proof:** The proof is by examining all cases.

The TB Rule affects the Degree-1 Rule either where it lengthens (by making topological or by adding a new topological segment) the arc incident to a degree-1 vertex, or where it creates a new degree-1 vertex. There is no reduction in the number of arcs operable by the Degree-1 Rule.

All of the other reduction rules operate on clean unique-in or locally unique-in arcs incident to sources (respectively unique-out or locally unique-out arcs incident to sinks); the s-t-s/t-s-t Rule additionally requires some locally unique-in/locally unique-out arcs that are not necessarily clean. Such arcs are not removed by the TB Rule. Further, it is obvious that a unique-in arc out of a source remains unique-in if other arcs incident to its head are removed. It is possible that such an arc could become part of a topological arc if the TB Rule removes all but one arc out of its head, however. Any TB arc removed affects at most one unique-in arc at its tail. (The symmetric argument holds for unique-out arcs into sinks.)

Recall that locally unique-in (respectively locally unique-out) arcs are incident to saddle vertices. Thus a locally unique-in arc $a$ remains locally unique-in: An arc adjacent in the cyclic order at the head of $a$ is marked both T and B only if the next arc in the cyclic order has the same

orientation. However, at each step of TB arc removal, the conflict resolution procedure does not allow the removal of two arcs with the same orientation that are adjacent in the cyclic order at some vertex. Finally note that if $a$ is clean it remains clean. If a pointer into its head were to be created across a face of which it lies on the boundary, an in-arc adjacent at the head would have to be removed. But the existence of such an adjacent arc would contradict the fact that this arc is locally unique-in. The arguments for locally unique-out arcs are symmetric.

Thus the application of the TB Rule does not conflict with any arcs operable by subsequent rules.

For Degree-1 Rule conflicts with subsequent rules, first note that removal of an arc by this rule does not make any clean arc dirty. Also note that the removal of such an arc changes the structure of only one face. Thus it is easy to see that at most one conflict can occur with the Adjacent Degree-2 Source and Sink Rule, and at most one with either the s-t-s or t-s-t Rule. Since a degree-1 arc is adjacent to at most one other source or sink at a saddle vertex, at most one application of the Consecutive Rule can be in conflict. A degree-1 arc can be adjacent to at most one index-1 saddle, so there can be at most three conflicts with the Index-1 Saddle Rule. It is obvious that any arc operable by the Unique-In/Unique-Out Arc Contraction Rule is unaffected by the removal of an arc by the Degree-1 Rule.

For the Unique-In/Unique-Out Arc Contraction Rule, two key observations are that contraction of such an arc never changes the face structure of the graph (e.g., top and bottom of every face stay the same), and that arcs incident to saddle vertices never are contracted. It is therefore easy to see that arcs operable by the Adjacent Degree-2 Source and Sink Rule, the Consecutive Rule, and the Index-1 Rule are not affected. Recall that the s-t-s/t-s-t Rule is only applied at sources or sinks that have either two or three locally unique-out (respectively locally unique-in) arcs; this plus the observation that the face structure is unchanged imply that there are no Unique-In/Unique-Out Arc Contraction Rule conflicts with s-t-s/t-s-t Rule applications.

Next consider the Adjacent Degree-2 Source and Sink Rule. Since the arcs incident to the source and sink involved are all removed, there is no conflict with the Consecutive Rule applied with either the source or sink as the center. This leaves at most four conflicts: a sink adjacent to the source in the cyclic order at either saddle vertex, or a source adjacent in the cyclic order to the sink at either saddle. For the s-t-s/t-s-t Rule there is a conflict if the sink might be involved in a t-s-t Rule application and the source in an s-t-s application (recall from the lemma statement that if an arc is operable by both rules it is not counted as a conflict). Since the s-t-s and t-s-t Rules apply to sources and sinks that share faces, and since the source and sink in question each have degree 2, there is at most one conflicting t-s-t Rule application involving the sink, and at most one s-t-s conflict involving the source. Finally, the source and sink each are adjacent to at most two vertices, so the number of Index-1 Saddle Rule applications affected is clearly bounded.

For the s-t-s/t-s-t Rule, first consider the Consecutive Rule. I give the argument for an application of the t-s-t Rule; the argument for an application of the s-t-s Rule is symmetric. There are two ways to conflict with a potential application of the Consecutive Rule: make one of the arcs involved "dirty" or remove one of the arcs involved. In applying the t-s-t Rule, no clean arcs are made dirty, so only the second case applies. The t-s-t Rule removes at most two arcs out of a source, so at most four potential applications of the Consecutive Rule are affected (again, I do not count cases in which the arc operable by a subsequent rule is operable by the current rule).

For t-s-t/s-t-s Rule conflicts with the Index-1 Saddle Rule, consider the case of an application of the t-s-t Rule. At most two arcs incident to the source are removed; the two sinks are combined. The only possible effects on potential Index-1 Saddle Rule applications are if the removed arcs are incident to index-1 saddles. No more than three conflicts are possible at each saddle. The argument for the s-t-s Rule is symmetric.

For Consecutive Rule conflicts with potential Index-1 Saddle Rule applications, note that the only way conflict can occur is through the removal of an arc incident to an index-1 saddle. Since a Consecutive Rule application removes exactly one arc that is incident to a source or sink, exactly one index-1 saddle can be affected. As in previous cases of conflict with the Index-1 Saddle Rule, there are at most three conflicts at that saddle.

Since the Index-1 Saddle Rule is applied last, there is nothing else to prove.

□

### 3.5.3 Proof of Main Lemma

It can now be shown that the reduction algorithm runs in a logarithmic number of iterations of the main loop.

**Lemma 3.5.3 [Main Lemma]** *For any embedded connected planar DAG consistent with the algorithm invariants, the generalized reduction algorithm works in $O(\log n)$ iterations of the main loop.*

**Proof:** This follows if I show that the reduction algorithm removes a constant proportion of the arcs in each pass through the main loop.

Consider the graph at the start of the main loop. After some application-specific processing (which does not change the graph), the graph is cleaned up. Cleanup leaves the number of vertices, sources, and sinks unchanged. The number of arcs does not increase; therefore it is sufficient to show that the algorithm removes a constant proportion of the arcs left after cleanup. Lemma 3.4.3 implies that a constant proportion of these remaining arcs are operable. All that is left is to show that the algorithm removes at least a constant proportion of the operable arcs.

To show this, I argue that the total number of operable arcs "knocked out" by conflicts (i.e., made inoperable) is bounded by some constant times the number of arcs removed. In most cases this is obvious because the total number of interrule and intrarule conflicts is bounded by a constant. The exception is TB arcs (the first Type II conflict graph does not have bounded degree). However, I have argued above that a constant fraction of such operable arcs are removed, which implies that the total number of arcs knocked out by intrarule conflicts is at most a constant times the number of TB arcs removed. Since the number of interrule conflicts with TB arcs is bounded by a constant, the result holds. Since every operable arc is either removed or is subject to a conflict with an arc that is removed, this implies at least a constant proportion of the operable arcs are removed.

□

## 3.6 Applications

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

### 3.6.1 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 an **initial set** of vertices in that DAG as the input, compute the set of vertices that are reachable via directed paths from the initial set. I refer to the vertices reachable in this way as the **solution set**; I include the initial set as a subset of the solution set. My solution to this problem consists of a set of application-specific actions taken at various points in the reduction algorithm; to show that it works I introduce invariants that allow me to prove that the result is correctly computed.

I introduce two flags at each vertex: a "reachable" flag indicating whether or not the vertex has been marked as reachable from one of the initial vertices, and an "active mark" flag that is used to determine whether or not 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. I use the term **correctly marked** to indicate that a vertex in the solution set has its "reachable" flag set, and that a vertex not in the solution set does not.

The basic reduction algorithm combines vertices as the graph is processed. The algorithm needs to keep track of such vertices while it computes reachability. Therefore I introduce the following terminology: A vertex in the current graph is an **original vertex** if it corresponds to exactly one of the vertices in the graph prior to the start of the reduction process (I consider sources added during preprocessing to be original vertices). The remaining vertices in the current graph correspond to two or more vertices that have been combined by various reduction rules; I refer to them as **combined vertices**. For each combined vertex I refer to the original vertices that have been combined into it as its **components**.

For the purpose of proving that the algorithm for the reachability application works, define the set of **active vertices**, which includes all original vertices that are not sources or sinks, plus any original sources that have active marks.

For the reachability application the algorithm keeps track of the status of each vertex (combined or original).

Define a **reduction propagation step** as follows:

- If a vertex $v$ is at the head of either a connectivity pointer or a directed arc that has an active mark at its tail, $v$ sets both of its flags (I say that the mark is propagated or passed over the arc or crosspointer). This rule also applies to internal vertices.
- If the directed arc over which a mark is passed is topological, all internal vertices of that arc are marked as reachable.
- If any internal vertex of a topological arc $a$ receives a mark, the "active mark" and "reachable" flags of the head of $a$ are both set.
- The "reachable" and "active mark" flags are unset for every sink and combined vertex.
- Any source that propagates an active mark unsets its "active mark" flag.

An **expansion propagation step** is defined similarly, except that all active vertices propagate their marks whether or not the "active mark" flag is set or not.

The following application-specific processing is added to the basic reduction algorithm:

- At the start of each cleanup phase $d$ propagation steps are performed, where $d$ is the degree limit introduced in Section 3.3.3. For each topological arc out of a source, if an active mark exists at an internal vertex higher than the high point, then the high point gets an active mark (this can be done in constant time in the CRCW model using the rank order on the topological arc). During the realignment phase, if a topological segment $seg$ of an arc out of a source is removed or replaced by a segment with no internal vertices, and if $seg$ contains a marked vertex, then the head of $seg$ is given an active mark (i.e., both flags are set).
- Whenever the TB Rule creates a topological arc $a$, if any internal vertex of $a$ has an active mark, the head of $a$ is given an active mark.
- Just prior to the application of specific rules, various numbers of propagation steps are done as follows:
  - One step is done before each of the Degree-1, Adjacent Degree-2 Source and Sink, s-t-s/t-s-t, and Consecutive Rules.
  - Two steps are done before the Unique-In/Unique-Out Contraction and Index-1 Saddle Rules.
  - For the TB Rule, one step is done prior to removing Type I TB arcs; two steps are done prior to removing each of Types II, III, and IV TB arcs.
- In rules where sources or sinks are combined with other vertices, the state of the vertices before combination is saved for the expansion phase, and the combined vertex is unmarked (i.e, neither of its flags are set).
- For the case of the Index-1 Saddle Rule in which the graph is split, the index-1 saddle vertex becomes a source. The active flag at this new source is unset.

Between the reduction and expansion phases, each topological arc that was removed marks itself according to any marks at any of its vertices. More specifically, all vertices beyond the first vertex marked reachable are marked reachable.

The application-specific steps added to the algorithm for the expansion phase are as follows:

- One expansion propagation step is done after arcs are restored for the Unique-In/Unique-Out Contraction and the Index-1 Saddle Rules; two are done for the Degree-1 Rule.
- As TB arcs are added back to the graph their internal vertices may need to be marked. This involves checking crosspointers and checking the tail of the arc. If the tail is marked reachable, all the internal vertices set themselves reachable. Otherwise, each internal vertex checks the lowest point that can reach it on each face it borders and sets its "reachable" flag accordingly. Also during expansion any vertices that became components of combined vertices are marked as necessary as they revert to original vertices. Note that the "active mark" flag is not used in this process. This step is done twice after Type IV TB arcs, twice after Type III TB arcs, twice after Type II TB arcs, and once after Type I TB arcs are restored.

At the end of the expansion phase the algorithm removes the restriction that sinks cannot be marked and does one more expansion propagation step to mark the sinks correctly.

It is easy to see that given the information on faces and topological arcs all of these application-specific actions can be done in constant time in the CRCW model.

The following lemma is useful in the proof that the reduction invariant holds through the cleanup phase (cleanup is discussed in Section 3.3.4, and some of the terminology used below is introduced there as well). A similar argument is used to show that the expansion invariant holds through the reverse of the cleanup phase during expansion. For simplicity, in the text below I refer to the highest points reachable from the frontier or beyond as "high points"; if no vertex is reachable from the frontier or below, then the frontier vertex is the high point.

**Lemma 3.6.1** *For each topological arc $a$ out of a source of degree $\leq d$, let $v$ be the the highest internal vertex on $a$ that both lies above the high point of $a$ and is reachable from an active mark. Then during reduction, after $d-1$ mark propagation steps $v$ is marked correctly.*

**Proof:** Such a vertex $v$ is reachable only from marks that lie above the high points for this source, so I can prove this claim by looking at the subgraph consisting of the source and all arcs (or segments of arcs) out to the high point, and all pointers that lie between two vertices in this subgraph. Note that there is such a $v$ for each arc in the subgraph that is reachable from an active mark. If $v$ is the source, the result is trivial. If $v$ already has an active mark, the result is again trivial. If the source does not have an active mark and $v$ is not yet marked, then the last link in the path from any mark must be a crosspointer. In particular, there must be a crosspointer from $v'$, the highest point reachable from a marked vertex on the other side of one of the adjacent faces: $v$ must be at the head of a crosspointer from some vertex $u$ reachable from a mark; if $u$ is not the highest reachable vertex on its arc $a'$, then the pointer rules indicate that the highest reachable vertex on $a'$ must have a crosspointer to a vertex on $a$ that is at least as high as $v$. Since such a crosspointer cannot point to a higher vertex than $v$ (that would contradict the fact that $v$ is the highest point reachable from a mark), the crosspointer must be to $v$.

Continue extending this path of crosspointers back until it reaches a marked vertex. Note that the path can never backtrack to an arc that has previously been visited: no higher point on such an arc can lie on such a path (this contradicts the fact that the path includes only the highest reachable vertices); no lower vertex or one already on the path lies on such a path because that would imply the existence of a cycle in the original graph, which is a DAG. Thus the path can have length at most $d-1$, and the phase of propagation across pointers causes the highest points reachable from marks to be marked.

□

To prove that the marking process specified above correctly marks the reachable vertices, I use the following invariants, one for the reduction phase and one for the expansion phase. The reduction invariant is as follows:

**Lemma 3.6.2** *During the reduction phase, the following two conditions hold:*

1. *There is no path from one active vertex to another through a vertex that is not in the active set (i.e., an original source with no active mark, a combined vertex, or a sink).*

2. *One or both of the following conditions hold for a vertex $v$ in the active set if and only if $v$ is in the solution set:*

- *$v$ is marked; or*
- *there exists a path of arcs or crosspointers from an active mark at an active vertex to $v$, and the vertices on this path are all active vertices.*

**Proof:** The proof proceeds by induction. The base case is the initial graph. The first part of the invariant is obviously true since all vertices are in the active set. The second part of the invariant holds by the definition of the problem (note that preprocessing adds only sources, so no added vertices violate the invariant).

For the induction step consider the effects of the mark propagation steps, cleanup, and applying each rule in a single pass through the main loop. By the induction hypothesis, the invariant holds at the start of a pass through the main loop; by the argument below, it holds at the end as well. □

**Cleanup:** The first cleanup phase is application-specific processing, which for the current application consists of $d$ rounds of mark propagation. Since mark propagation does not change any path in the graph or combine any vertices, the first part of the invariant obviously remains true.

It is also obvious that the second part of the invariant continues to hold because it holds prior to propagation by the induction hypothesis, and because marks are propagated only over paths of arcs and crosspointers through active vertices.

At this point any sources or sinks with degree higher than $d$ (the cleanup degree constant) drop out of the cleanup process. Since they are unaltered by further cleanup steps, no changes to the invariant occur. Only sources and sinks that are cleaned up need be considered.

The determination of the highest internal vertex on an arc out of a source reachable from the frontier (respectively lowest internal vertex on an arc into a sink that can reach the frontier) does not affect the invariant.

To show that the invariant holds after realignment, first note that if $v$ is an active vertex that lies between a high point and its cleaned source or below a low point and its cleaned sink, then $v$ is removed. Second, it is straightforward to see that there is a path between two vertices after realignment only if there was a path between those vertices prior to realignment. In conjunction with the induction hypothesis and the previous arguments, this implies both that the first part of the invariant continues to hold, and that there is no path from an active mark to any active vertex not in the solution set (i.e., the second part of the invariant holds for vertices not in the solution set). Third, the second part of the invariant continues to hold for any marked active vertex. All that is left to show is that the second part of the invariant continues to to hold for unmarked vertices in the solution set.

Consider any path $P$ from an active mark to a remaining unmarked active vertex such that $P$ exists prior to realignment. Since the realignment actions do not disturb paths that do not include any vertices above high points or below low points, if $P$ is such a path it remains after realignment. By definition of low point, once a path reaches a vertex below the low point on an arc into some sink, all subsequent vertices on that path must be below the low point on some arc into that sink. Thus, $P$ cannot pass through a vertex below the low point at any sink (recall that no vertices below low points remain after cleanup). The only remaining case to consider is if $P$ passes through vertices above the high point at some source. By the definition of high point, such a path cannot include a vertex above a high point on an arc out of some source unless the path starts at such a

vertex at that source. Thus if $P$ is such a path it starts at an active mark above a high point at some source. Assume that this is the case. I need to show that any vertex on $P$ that remains after realignment remains reachable from an active mark.

Note that Lemma 3.6.1 above implies that any high point $h$ reachable by such an active mark at the same source gets an active mark as a result of the application-specific processing:

- either some vertex above $h$ is reachable by such a mark, in which case the lemma shows that the highest reachable point above $h$ is marked, which implies $h$ will be marked when marks are propagated to high points,
- or $h$ is the highest point on its arc $a$ reachable by such a mark. In this case the last link on the path from the mark to $h$ must be a crosspointer from some vertex on an arc $a'$. But then the crosspointer from $u$, the highest reachable point above the high point on $a'$, must also have $h$ as its head (it must point at least as high as $h$, but no higher point on $a$ is reachable from such a mark). By the lemma, $u$ has been marked after $d-1$ propagation steps; then $h$ has been marked after the $d$ propagation steps.

Thus the second part of the invariant holds for paths that go through high points.

If $P$ does not go through a high point, there must be a first vertex $v$ on the path that lies below a high point, and the path must follow a crosspointer from a vertex $u$ above a high point to $v$. But this implies that there is a crosspointer $p$ from $u'$, the highest point marked on $u$'s arc, to some point $v'$ at or above $v$ on $v$'s side of the flow face. If $v'$ is above the high point, the high point is marked as per the previous paragraph. Otherwise Lemma 3.6.1 says that $u'$ is marked by the time $d-1$ propagation steps have occurred, so $v'$ has been marked by the time $d$ propagation steps have occurred. Either way, the claim holds.

**TB Rules:** The TB Rule does not combine vertices or create new paths, so the first part of the invariant continues to hold.

To show that the second part of the invariant continues to hold, I show that it holds after each step in the rule application/conflict resolution procedure.

The rule is first applied for Type I TB arcs. Note that the conflict resolution for this step ensures that at most one arc per face is removed in this step. First consider the paths left after Type I TB arcs are removed. In particular, I want to show that for any pair of active vertices $u$ and $v$ that remain after Type I arcs are removed, if there was a path $P$ from $u$ to $v$ prior to removal then there is a path $P'$ from $u$ to $v$ after removal. Since the first part of the invariant holds at the time of removal, the path left after removal includes only active vertices. There are four cases to consider on the basis of how a removed Type I arc $a$ is involved in the original path $P$:

- The path includes $a$. There are two possibilities: First, $a$ may be replaced by a crosspointer, which replaces $a$ in the path. Second, the tail of $a$ may already have a crosspointer to a point above $a$'s head on the other side of the face. In this case $a$ is not replaced by a crosspointer. However, there is a path from the tail of $a$ to the head of $a$ across the crosspointer and down the opposite side of the face. This path was in existence prior to the removal of $a$. Since the second half of the invariant held previously, and since the head and tail of $a$ are active, all vertices on this path must be active. Since no other arcs on this face are removed, this path is not broken by the removal of any other Type I TB arc.

- The path enters the tail of $a$ and leaves $a$ via a crosspointer out of an internal vertex across face $f$. In this case it is necessary to consider whether $a$ is marked T or B on $f$ (recall that Type I TB arcs are marked T on one adjacent face and B on the other, and that they are not both T and B on any face). If it is marked T, then the tail of $a$ is the top of the face and there is a path from the tail of $a$ to the head of the crosspointer along the opposite side of the face prior to $a$'s removal. Since at most one arc per face is removed, this path is not affected by Type I arc removal. If it is marked B, the crosspointer at the tail of $a$ points to a vertex on the opposite face as high or higher than the crosspointer involved in the original path. Thus using the crosspointer at the tail of $a$ and part of the opposite side of the face boundary gives an alternative path; again, this path is not affected by Type I arc removal.

- The path enters an internal vertex of $a$ via a cross pointer across face $f$ and leaves via $a$'s head. Let $w$ be the internal vertex on $a$ where the crosspointer enters. Again, consider the cases in which $a$ is marked T or B with respect to $f$. If it is marked T, then the crosspointer out of $w$ on the other face $f'$ adjacent to $a$ reaches a point at or above the head of $a$ ($a$ is marked B with respect to $f'$); this provides that alternative path and is not affected by other Type I arc removals. If $a$ is marked B with respect to $f$, then there is a path from the tail of the crosspointer to the head of $a$ along the side of $f$ opposite to $a$. As in previous cases, this path is not affected by removal of any other Type I TB arc.

- The path enters an internal vertex $w$ of $a$ via a crosspointer $p$ across face $f$ and leaves $a$ via a crosspointer $p'$ across face $f'$ out of internal vertex $w'$. By the specification of the crosspointers, the crosspointer $p''$ out of $w$ across $f'$ reaches a point on the opposite side of $f'$ that is as high or higher than the point reached by $p'$. Thus after arc removal the crosspointer that results from splicing $p$ and $p''$ and possibly a segment of what was the opposite face of $f'$ provides the alternative path.

The only other problem that could occur during Type I arc removal is that an active mark at an internal vertex might be deleted when the associated arc is removed. It is necessary to show that this does not affect the invariant by leaving some unmarked active vertex in the solution set without a path from an active mark. I show that this is prevented by the single step of mark propagation is done prior to Type I arc removal.

To see that any vertex reachable from an active mark is still reachable after the Type I TB arcs are removed, consider the situation just after removal. Any active mark removed must be at an internal vertex $v$ of some topological arc $a$. There are two cases to consider. The first case is that the mark started at $v$. In this case the propagation rules insure that if the head is an active vertex, it is marked with an active mark, which, given the argument above, implies that the invariant continues to hold for any path from $v$ through the head of the arc. The other paths out of $v$ are via crosspointers. Note that the pointers out of $v$ reach as high or higher than the crosspointers out of vertices lower than $v$ on $a$. Thus for any path out of a lower vertex it is possible to find a path out of a crosspointer at $v$ and down the opposite side of some face; it is only necessary to consider the crosspointers out of $v$. The heads of $v$'s crosspointers are marked by the propagation phase, and since they lie on a face common to $a$, they are not removed when the Type I TB arcs are.

The second case to consider is when $v$ receives an active mark as the result of the propagation

phase. If the mark propagates in via the tail of $a$, then $a$'s head is marked. Furthermore, since a source cannot be at the tail of a Type I TB arc, the active mark remains at the tail. Thus, any remaining vertex that was on the boundary of the face $f$ for which $a$ is marked T is reachable from this active mark. The only paths left to consider are those that leave $v$ through a crosspointer on $f'$, the face for which $a$ is marked B. But the crosspointer out of $a$'s tail reaches as high as the crosspointer out of $v$ and provides a path from the active mark at $a$'s tail to any vertex on the opposite side of $f'$ reachable from $v$. Since no other arcs on $f$ or $f'$ are removed, the claim holds. If $v$ received the active mark across a crosspointer $p$, then again the head of $a$ is marked. If $p$ is across face $f$, $v$'s crosspointer on $f$ is to a point below the tail of $p$ since the graph is a DAG. Thus the only paths left to worry about are those that cross a second face $f'$ of which $a$ is on the boundary. Such paths are via a crosspointer $p'$ out of $v$. However, $p$ and $p'$ are spliced into a new pointer that provides a path from the tail of $p$ (where the active mark remains) to the head of $p'$.

The invariant therefore holds after Type I TB arc removal. Next consider the situation when Type II, Type III, and Type IV TB arcs are removed. These cases are similar and can be treated together.

Start with the useful observation that an arc of these types is marked both T and B on at least one face $f$, and that the conflict rules assure that no arcs are removed from the opposite side of $f$. Thus, when such an arc is removed a path from the tail to the head remains undisturbed.

I again need to show that if a path from a vertex $u$ to a vertex $v$ exists before TB arcs of any of these types are removed, then it exists after the arcs are removed. The arguments to show this are similar to those used for Type I TB arcs. However, in this case there is the added complication that two such arcs connected by a crosspointer can be removed simultaneously. As a result, there are more cases to consider when an arc $a$ is removed:

- The path enters the tail of $a$ and exits the head.
- The path enters the tail of $a$ and exits a crosspointer to a vertex that is not removed.
- The path enters $a$ via a crosspointer from a vertex that is not removed and exits via the head.
- The path enters $a$ via a crosspointer from a vertex that is not removed and exits via a crosspointer to a vertex that is not removed.
- The path enters the tail of $a$ and exits a crosspointer to a vertex on $b$, and exits via the head of $b$.
- The path enters the tail of $a$ and exits a crosspointer to a vertex on $b$, and exits $b$ via a crosspointer to a vertex that is not removed.
- The path enters $a$ via a crosspointer from a vertex that is not removed, exits via a crosspointer to a vertex on $b$, and exits the head of $b$.
- The path enters $a$ via a crosspointer from a vertex that is not removed, exits via a crosspointer to a vertex on $b$, and exits $b$ via a crosspointer to a vertex that is not removed.

The details of the arguments for these cases are similar to those for the Type I TB arc arguments, and are left to the reader.

The arguments that no active marks are lost are also similar to the arguments in the Type I TB arc case. There may be one additional crosspointer to deal with; however, the two propagation steps for these three types are sufficient to insure that the marks reach vertices that are not removed. Once again the details are left to the reader.

This proves the claim for the TB Rule.

**Degree-1 Rule:** Once again note that application of this rule does not create any new paths, so by the induction hypothesis and the preceding arguments, the first part of the invariant continues to hold. Also, no paths from active marks to vertices not in the solution set exist after rule application, so the second part of the invariant continues to hold for active vertices not in the solution set.

To show that the second part of the invariant continues to hold for active vertices in the solution set, first consider a degree-1 source. Note that one cannot assume that $a_s$, the arc out, is clean, because the application of the TB rules may have made the arc out a longer topological arc. Because all faces are flow faces, $a_s$ is both the left path and the right path of the boundary of a flow face, and is the top arc on both sides of the face. Thus there are no pointers into $a_s$ from any vertex outside it; such a pointer would be a backpointer and would imply that the input DAG had a cycle, which is impossible. Thus the only paths between active vertices that include internal vertices on $a_s$ or its head are those that start at such a vertex. Therefore if there is no mark internal to the removed arc, the invariant continues to hold; if there is an active mark internal to the arc then any vertex reachable from the mark is reachable via a path through the head of the arc, which either gets an active mark as a result of the propagation step for this rule application, or has one already.

The case for a sink is simple. The arc $a_t$ into a degree-1 sink is the bottom arc on both sides of a flow face. Therefore there can be no paths out of any vertices on $a_t$ to higher points on the face because the graph started as a DAG. This implies that these vertices are not included in any path between active vertices that remain after this rule application, nor does any such path start at one of these vertices. The arc $a_t$ can be removed without affecting paths from active marks to remaining active vertices.

**Unique-In(Unique-Out) Arc Contraction Rule:** Note that a necessary condition for arc contraction to change the connectivity of the graph is that there be a path out of some point on the arc at or below its tail, and a path into some point on the arc at or above its head. If the arc is not topological, this translates to a path into the head of the arc and a path out of the tail of the arc.

First consider a unique-in arc $a$ incident to a source $s$. Since this rule is only applied if $a$ is clean, it is easy to show that the contraction of $a$ does not change the connectivity of the graph: Because $a$ is clean, there are no pointers into $a$ or its head across the faces of which $a$ is on the boundary, and, since $a$ is the unique arc into its head, any pointers across any other faces into the head of $a$ would be backpointers and contradict the fact that the graph is a DAG. By the condition stated above, contraction of $a$ cannot create any new paths, so the first part of the invariant and the invariant condition on vertices not in the solution set continue to hold. It is also easy to see that if $P$ is any path from an active mark to an active vertex such that neither $s$ nor the head of $a$ lie on $P$, then $P$ is unaffected by the contraction. The only cases left are if either $s$ or the head of $a$ had an active mark prior to contraction. These cases are handled by the two propagation steps prior to rule application, which either mark the vertices reachable from these vertices or leave an active mark at an intermediate active vertex that is not affected by the application of this rule.

For sinks, a symmetric argument shows that no new paths are added, which implies that the first

part of the invariant and the invariant condition on active vertices not in the solution set continue to hold. Also, any path that does not pass through the tail of the contracted arc is unaffected (the sink is never an active vertex and has no paths through it). Since the only paths through the tail of $a$ must next cross $a$ and terminate at the sink, the condition on active vertices in the solution set is unaffected.

**Adjacent Degree-2 Sources and Sinks Rule:** For this rule the algorithm removes a source and a sink and their (clean) incident arcs. No new paths are created, so the first part of the invariant continues to hold, and the second part of the invariant continues to hold for active vertices not in the solution set. I also need to show no paths from active marks to active vertices are broken. The only such paths that can be broken are those that start at an active mark at the removed source, so the propagation step insures that any vertex reachable from a mark at the source is either marked or reachable from an active mark at an intermediate vertex along the original path.

**Source-Sink-Source (s-t-s)/Sink-Source-Sink (t-s-t) Rule:** First consider the s-t-s rule. Two sources get combined into a single vertex, which is not active. The only new paths created are those that start at one or the other of these sources, so no paths are created that violate the invariant condition for active vertices not in the solution set. Since there is a propagation step, neither of these sources remains active and the first part of the invariant continues to hold. It is also necessary to worry about breaking paths from active marks to active vertices in the solution set to complete the argument that the invariant holds for applications of this rule. But the only such paths affected by this rule are those that start at the sources (the sink is not in the active set, so removal of an arc into it does not break any such paths). The propagation step arguments used for previous rules apply here and give the desired result.

For the t-s-t Rule, two sinks get combined into a single vertex, which is not active. No new paths are created to any active vertex, so no paths are created that violate the invariant condition for active vertices not in the solution set, nor are any created that violate the first part of the invariant. I again only need to worry about breaking paths from active marks to active vertices in the solution set to complete the argument that the invariant holds for applications of this rule. But the only such paths that are affected by this rule are those that start at the source that loses an arc. The propagation step arguments used previously again apply and give the desired result.

**Consecutive Rule:** The arguments here are essentially the same as those for the s-t-s and t-s-t Rules: a clean arc is deleted and two sources or sinks are combined. No new paths are created, so the first part of the invariant is unaffected, as is the condition on vertices not in the solution set. No paths from active vertices to other active vertices are affected with the exception of paths from active marks at sources involved in the rule application; as in previous cases, the propagation step for this rule insures that the invariant still holds for active vertices in the solution set.

**Index-1 Saddle Rule:** There are two basic cases to consider: application with respect to sources and application with respect to sinks. In the situation where the rule is applied with respect to sources, there are two subcases: applications that contract arcs, and applications that separate the graph.

First consider the case of two distinct sources $s_1$ and $s_2$ with clean arcs into the saddle. By the same kind of arguments used in the Unique-In/Unique-Out Arc Contraction Rule, the only paths into the saddle are the two arcs out of the sources. If both arcs are contracted by the rule application, the same argument applies as for the Contraction Rule; if only one arc is contracted

(w.l.o.g. assume the arc incident to $s_1$), then the only new paths created are those starting at $s_2$ and exiting the combined vertex via an arc that was out of $s_1$. However, since two propagation phases were performed prior to contracting the arc, $s_2$ no longer has an active mark (remember that the "active mark" flag at a source is unset after propagation) and thus is not an active vertex, so the invariant is not violated. The rest of the argument proceeds as for the Contraction Rule.

Next consider the case where the Index-1 Saddle Rule is applied at a source that is incident to the only two arcs into the saddle. In this case the two arcs from the source to the saddle are deleted and the graph is separated into two graphs. First consider the case for a degree-2 source. By previous arguments used for other rules in which arcs were only deleted, the first part of the invariant and the second part of the invariant's condition on active vertices not in the solution set continues to hold since no new paths are created. If the source has degree 3, the tail of the third arc out of the source becomes the former saddle, thus creating new paths. However, the former saddle becomes a source and (as part of the application specific processing) loses any active mark. Thus the new paths do not violate either the first part of the invariant or the second part of the invariant's condition on active vertices not in the solution set, which continue to hold. The rest of the argument is the same in the case of either degree-2 or degree-3 sources: For active vertices in the solution set I need to show that no paths from active marks to unmarked vertices are broken. Since the only paths broken by splitting the graph are those that go through the saddle, and since the only arcs into the saddle are from the source, I only need to worry about the case in which the source has an active mark. However, by the same arguments used above, the two propagation steps prior to rule application insure that the invariant continues to apply for active vertices in the solution set, which are either marked or are reachable from some intermediate active vertex not affected by the rule application.

Third, consider the case of two distinct sinks $t_1$ and $t_2$ with clean arcs out of the saddle. By arguments used for the Unique-In/Unique-Out Arc Contraction Rule, the only paths out of the saddle are into the sinks. If both arcs are contracted, no new paths are added, which implies that the first part of the invariant and the invariant condition on active vertices not in the solution set continue to hold. If only one arc is contracted, the only new paths created are those that extend a path into the combined vertex created from the contracted sink (say $t_1$) and the saddle. These paths all either end at the combined vertex or at $t_2$. Since sinks and combined vertices are not in the active set, the the first part of the invariant and the condition on active vertices not in the solution set are again unaffected. For active vertices in the solution set, no paths are broken and the second part of the invariant continues to hold.

Finally, consider the case where one sink is incident to the two clean arcs out of the saddle. In this case the two arcs from the saddle to the sink are deleted and the graph is separated into two graphs. First consider the case for a degree-2 sink. By previous arguments used for other rules in which arcs were only deleted, the first part of the invariant and the invariant condition on active vertices not in the solution set continue to hold since no new paths between active vertices are created. If the sink has degree 3, the head of the third arc into the sink becomes the former saddle, thus creating new paths. However, the former saddle becomes a sink, which is not in the active set and will not be marked as part of the reduction application-specific processing. Thus the new paths do not violate either the first part of the invariant or the second part of the invariant's condition on active vertices not in the solution set, which continue to hold. The rest of the argument is the

same in the case of either degree-2 or degree-3 sinks: For active vertices in the solution set I need to show that no paths from active marks to unmarked vertices are broken. Since the only paths broken by splitting the graph are those that go through the saddle, and since the only arcs out of the saddle are to the sink, it is clear that for active vertices in the solution set, no paths are broken and the invariant continues to hold.

□

At the end of the reduction phase, consider the active vertices. If reduction stops when the graph is some constant size, all active vertices reachable from an active mark can then be marked in constant time.

As noted above, between the reduction and expansion phases all removed topological arcs are correctly marked by propagation of any marks on internal vertices. Since the rank order of the vertices on the topological arc is known at the time of removal, standard techniques can be applied to determine the first marked internal vertex in constant time in the CRCW model. All vertices can read this rank and mark themselves if they have a higher rank. Thus all this processing can be done in constant time in the CRCW model.

At this point the expansion phase begins. Expansion proceeds by reversing the steps of the basic reduction algorithm, with application-specific steps added as specified above. The moving of vertices between various sets for analysis purposes is also be reversed. Recall that during the expansion phase all marks at original vertices are allowed to propagate.

The expansion invariant is as follows:

**Lemma 3.6.3** *During the expansion phase, all active vertices are correctly marked.*

**Proof:** This proof also works by induction on backward passes through the main loop. The base case follows from the discussion above and the following observations about the reduction procedure: First, since there are no active marks at vertices not in the active set, and since all mark propagation is from vertices with active marks to active vertices, the reduction invariant therefore implies that no active vertices are incorrectly marked.

Note that for the Source-Sink-Source (s-t-s)/Sink-Source-Sink (t-s-t) Rule, the Consecutive Rule, and the Adjacent Degree-2 Sources and Sinks Rule, the only change to the active set when these rules were applied in the reduction phase was that some sources dropped from the active set. In particular, sources that had an active mark propagated that mark out. By the argument above, these vertices were correctly marked prior to rule application, and thus are marked correctly when they are returned to the set of active vertices. Therefore, in reversing these steps the expansion invariant remains unchanged.

For the Unique-In(Unique-Out) Arc Contraction Rule and the Index-1 Saddle Rule, note that the only vertices that could be dropped from the active set when these rules are applied are those at the head (respectively tail) of an arc contracted into a source (respectively sink), plus any active source involved in the contraction. (In the case of the Index-1 Rule, these arcs may have been removed rather than contracted).

Start by considering the case where an arc $a$ incident to a source was contracted. In expanding, if the source becomes active, then by the definition of the active set it must have had an active mark prior to contraction, and is thus marked. Because the reduction invariant held at the time of contraction this mark is correct. If $v$, the head of $a$, becomes active, then since the reduction invariant held prior to contraction either $v$ is marked correctly or there was an active mark at some

vertex with a path (through active vertices) to $v$. But I proved above that $v$ is reachable only from the one or two sources that have arcs into it, in which case if $v$ was in the solution set but was unmarked prior to rule application, then there must have been an active mark at such a source. This mark would have marked $v$ during the propagation step for this rule.

In the case of an arc $a$ with tail $v$ that was contracted with a sink $t$, recall that sinks are never in the active set, and thus it is only necessary to consider the case when $v$ becomes active upon rule reversal. First consider the case when $v$ is not in the solution set. Since the reduction invariant held prior to contraction, $v$ is not marked, nor are any active vertices that have paths to $v$. Therefore after the expansion propagation step $v$ is still unmarked. If $v$ is in the solution set, either $v$ was marked prior to contraction, or there was an active mark at some vertex $u$ and a path form $u$ to $v$ through active vertices. This implies that prior to contraction there was an active vertex $w$ with an arc (or perhaps a pointer) into $v$. By the induction hypothesis and the fact that $w$ was not combined during this rule (i.e., it remained active), $w$ is correctly marked when $a$ is restored, so $v$ is correctly marked during the subsequent expansion propagation step.

For cases in which the Index-1 Rule separates the graph, first consider the case of a source incident to both arcs into an index-1 saddle $v$. If after restoring these arcs $v$ is active, the following cases could occur:

- $v$ is in the solution set and is already marked. The invariant obviously holds here.
- $v$ is not in the solution set. Note that the only paths into $v$ are the arcs from the source. Then the reduction invariant implies the sources cannot be marked. Thus $v$ is not marked by the propagation step and the invariant holds.
- $v$ is in the solution set and not marked. By the reduction invariant that held prior to rule application, the source must have been marked and the propagation step prior to rule application would have marked $v$, so this case does not occur.

Now consider the case of a sink incident to both out arcs from an index-1 saddle $v$. Each copy of $v$ (one in each of the graphs left after separation) became a sink and was subsequently inactive. If $v$ becomes active when the rule is reversed, the situation is basically the same as in the case of expansion of a unique-out arc incident to a sink. In this case if $v$ is in the solution set but unmarked, it is guaranteed that an adjacent vertex remained active when this rule was applied (this follows from the conflict resolution procedure for the Index-1 Rule and the two steps of propagation done prior to applying this rule in the reduction process), and is now marked. This insures that $v$ is marked during the expansion propagation step.

**Degree-1 Rule:** When a degree-1 arc is restored, vertices internal to that arc may become active. Consider such an arc $a$. As was noted above in the argument for the reduction invariant, there are no crosspointers into the internal vertices on $a$. To see that the invariant continues to hold, consider the following sets of vertices that become active:

- Vertices in the solution set that are marked. It is obvious that the invariant holds for these vertices.
- Vertices in the solution set that are unmarked. Note that if these are internal vertices, at the time of their removal the only path to them was through a higher vertex on the arc.

If $a$ is incident to a source, then the reduction invariant implies that there must be a mark somewhere on $a$ that was propagated to all reachable vertices in the step between expansion and reduction. If $a$ is incident to a sink, then either the situation described above occurred, or the tail of $a$ was active. In this case there is a marked vertex with an arc or pointer to the tail of $a$, and the two expansion propagation steps insure that the vertices of this type are marked (the tail becomes a sink when $a$ is removed, so two steps are necessary).

- Vertices that are not in the solution set. Recall that the reduction invariant implies that these vertices are not marked, no higher vertex on the arc can be marked, and no vertex incident to an arc into any vertex on $a$ can have a mark. Thus these vertices remain unmarked.

**TB Rules:** As in previous cases, the reduction invariant allows one to argue that vertices not in the solution set are not marked. Thus one need only consider restored active vertices that are in the solution set. Such vertices that are already marked are consistent with the invariant, so I only need to consider unmarked restored vertices in the solution set.

In the expansion phase the algorithm may restore a topological arc with internal vertices in the solution set that were not marked at the time of removal. To see that these vertices are properly marked after the propagation step following the arc's restoration, first note that any marks at internal vertices were propagated correctly between the reduction and expansion phases. Thus I only need to consider marks that come from outside the arc. By the reduction invariant, the only paths through which such marks can reach the arc must be through active vertices either at the tail of this arc or at the tails of pointers incident to internal vertices on the arc (as noted above, the marks could be at the tail of a path of pointers through internal vertices on restored arcs, but such paths can have length two at most). By the induction hypothesis these active vertices are correctly marked, so in the case where there is a mark at the tail of some restored arc, the internal vertices are marked correctly by the expansion propagation step.

Now consider the case in which there is an unmarked vertex $v$ on restored arc $a$ such that $v$ is in the solution set and the tail of $a$ is unmarked. As noted above, there must be a path from some marked vertex that first crosses a crosspointer into $a$, then travels along some (possibly empty) segment of $a$ to $v$. I now use the following fact, which is easy to prove: There is a marked vertex $u$ on the opposite side of a face $f$ from $v$ such that there is a path from $u$ to $v$ of the form described above if and only if the lowest vertex across $f$ that can reach $v$ is marked. Therefore the marking process described in the application-specific processing works (note that the restoration of Type II TB arcs can break pointers into three pieces, so two steps are necessary in that case).

**Cleanup:** In reversing the cleanup process, the algorithm can restore some vertices that lie above high points or below low points to the active set. As in previous cases, vertices not in the solution set are not marked, nor will they be marked upon restoration. Thus I again only need worry about vertices in the solution set. If such vertices are marked, the reduction invariant implies they are correctly marked; thus I only need to worry about unmarked vertices in the solution set that become active.

For such vertices above high points, I argued above that the highest such points on each topological arc at a cleaned source were marked prior to removal. This implies that all lower vertices on the topological were marked in the step between reduction and expansion, so no unmarked vertices of the type I am considering remain above high points.

Thus I need only to show that unmarked vertices below low points and in the solution set are correctly marked after the $d$ expansion propagation steps following the restoration of the previous graph structure. The argument is similar to that in Lemma 3.6.1. The claim follows by noting that for each arc containing a vertex that should be marked, there is a highest point at or below the low point that should be marked. If this highest point is already marked, the mark can be propagated along the topological arc to mark every active vertex on the arc as described below. If the highest point $v$ reachable by a mark is not yet marked, then the last link in the path from a mark must be over a crosspointer. In particular, there must be a crosspointer from the highest point reachable from a marked vertex on the other side of one of the adjacent faces: there must be a crosspointer from a vertex $u$ reachable from a mark; if $u$ is not the highest reachable vertex on its arc, then the pointer rules indicate that the highest reachable vertex on $u$'s arc must have a crosspointer to a vertex on $v$'s arc that is at least as high as $v$. Since such a crosspointer could not be to a higher point than $v$ (that would contradict the fact that $v$ is the highest point reachable from a mark), the crosspointer must be to $v$.

Continue extending this path back until it reaches a marked vertex. Note that the path can never backtrack to an arc that has previously been visited: no higher point on such an arc can lie on such a path (this contradicts the fact that the path includes only the highest reachable vertices); no lower vertex or one already on the path could lie on such a path because that would imply the existence of a cycle in the original graph, which is a DAG. There are two cases to consider. First, the path works its way back to some marked vertex at or below the low point. Since no arc can appear in the path more than once, the path has length at most $d$, and the phase of propagation across pointers causes the highest points reachable from marks to be marked. The propagation of marks along topological arcs marks the rest of the vertices on the arc. If the last vertex at or below a low point is not marked, then the path from a mark must go higher than the low point through some active vertex. In particular the path from a mark into this last vertex must be a crosspointer; the vertex at the tail of the crosspointer must be active (otherwise the reduction invariant would have been contradicted) and thus must be marked by the induction hypothesis. The low point on bottom arc of the side of the face below the marked vertex is not on this path by the construction since it is marked or not active; thus the path is again of length at most $d$, and is marked by the propagation steps.

To see that it is possible to propagate marks at internal vertices along topological arcs, recall that the algorithm keeps a rank ordering of the vertices on the topological arc. Thus, in the CRCW ARBITRARY model standard techniques can be applied to determine in constant time the highest marked internal vertex, and therefore in constant time every lower vertex can mark itself.

□

These invariants are sufficient to prove that at the completion of the algorithm the graph is correctly marked.

**Theorem 3.6.4** *The marking procedure specified above solves the many-source reachability problem.*

**Proof:** By Lemma 3.6.3, every vertex in the graph is correctly marked at the end of the expansion phase except sinks. Thus, the last step of marking sinks marks each sink if and only if it is in the solution set.

□

### 3.6.2 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 [Kao93] the running time becomes $O(\log^3 n)$.

## 3.7 Planar DAG Reduction: Summary and Contributions

In this chapter I have presented the Poincaré index formula, a combinatorial property of embedded planar directed graphs, and have shown that it can be applied in the design and analysis of parallel algorithms.

I have also shown a reduction scheme for planar DAGs that allows such a graph to be reduced to constant size and expanded back. This reduction process allows the overlaying of applications in a fashion similar to parallel tree contraction. I have demonstrated such an application and shown that computing multi-source 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 conjunction with the strongly-connected components algorithm of Kao [Kao93] it is possible to compute multiple-source reachability for general planar digraphs in $O(\log^3 n)$ time using $O(n)$ processors, an improvement over previous results.

# Chapter 4

# Open Problems and Future Work

Both Chapters 2 and 3 present results suggesting a broad range of possibilities for further work. I discuss some of these possibilities in this chapter, starting with extensions to the work on spectral partitioning.

## 4.1 Further Work on Spectral Separator Algorithms

There are many ways to build on the Laplacian spectrum work, including the following:

- finding ways to improve current spectral partitioning algorithms;
- further analysis of Laplacian spectra, and of the partitions produced by spectral separator algorithms for various graph classes; and
- extending the techniques developed in Chapter 2 to solve additional problems.

The first two are closely intertwined, so I discuss them together.

An interesting question is why the spectral algorithms considered in Chapter 2 do poorly on certain graphs. Intuitively, the orientation of $\mathbf{u}_2$, the second smallest eigenvalue, reflects two properties of a graph: $\mathbf{u}_2$ tends to lie along long paths and across bottlenecks. If it lies across a bottleneck, the spectral algorithms tend to give good separators; if it lies along a long path, things can go wrong. This latter case occurs for the tree-cross-path graph; in that case, the bottleneck does not coincide with the long path. Note, however, that all the graphs I consider have some eigenvector that gives a good cut when the "best threshold cut" algorithm is applied to that vector.

This suggests the following question: Is there always an eigenvector that gives (via the "best threshold cut" heuristic) a separator with a cut quotient within a reasonable factor (say, $O(\log n)$) of the isoperimetric number? If so, is there a means for computing this eigenvector efficiently? If not, what does a counterexample look like?

Assuming for the moment that such an eigenvector does exist, it may have some particular form. To understand one such possible form, one ought to read Fiedler's fundamental paper [Fie75] in which he proves the following beautiful property of $\mathbf{u}_2$: the vertices that receive value 0 or greater from the eigenvector form a connected component; likewise, the vertices that receive value 0 or

less form a connected component. Other eigenvectors can have this property, which I call the Fiedler property. I refer to such vectors as **Fiedlerian**; while it would be nice to call them "Fiedler vectors", that term is already in common use to denote $\mathbf{u}_2$. Another possibility is that only certain graphs have Fiedlerian vectors that provide good cuts. The crossproduct examples I consider above do; is there a general "crossproduct-like" class for which this is true?

Even if every Laplacian has a Fiedlerian vector that gives a good cut for the associated graph, the question of how to find Fiedlerian vectors efficiently remains open.

Another way to approach the analysis of Laplacian spectra is to develop results about specific classes of graphs. For example, I have shown that spectral bisection does quite poorly for bounded-degree planar graphs; however, the counterexamples for the other algorithms are all non-planar. This raises the question of how well the "best threshold cut" algorithm does for planar graphs.

There are many interesting related questions: Are there ways to classify graphs that include classes for which the spectral methods considered in Chapter 2 provide good cuts? It would be interesting to categorize which spectral algorithms work well for which sorts of graphs; this may require the definition of new properties for classifying graphs.

Another potential way to improve the performance of spectral separator algorithms is to use information about the graph's structure. The algorithms use only eigenvector values; they do not consider structural properties of the graph. It may be possible to process some additional information (e.g., geometrical information) to improve the quality of the separators produced.

It is also interesting to consider if there are implications for other spectral methods in this work. For example, Bruce Hendrickson of Sandia National Laboratories has noted that the roach graph is a counterexample for spectral algorithms intended to produce small-profile orderings of sparse matrices.

Another area for work is extending the results and techniques in this section to solve additional problems. For example, questions about the results from Chapter 2 have led to a technique for generating lower bounds on $\lambda_2$ based on embedding the clique into the graph of interest. That work in turn has led to work on determining how far such lower bounds can be from the actual $\lambda_2$. Results will be documented in a future paper.

The lower bound work grew out of comments by practitioners that the double tree was not the best graph to use in my counterexample crossproducts if I wanted graphs like those arising in practice. Replacing the complete binary trees with square grids was suggested. The resulting graph is a crossproduct between a path graph and the double grid, pictured in Figure 4.1 below. I call the resulting graph the "twin towers" graph based on its resemblance to the World Trade Center towers in New York. The bounding technique is necessary in bounding $\lambda_2$ of the double grid within a constant. The subsequent analysis of the twin towers graph is similar to the analysis for the tree-cross-path graph. The result for the "best threshold cut" algorithm, however, no longer has the ratio of $q_{min}$ to $i(G)$ as large as possible; instead, the ratio is less by a factor logarithmic in the size of the graph. Still, the spectral cut in this case is poor.

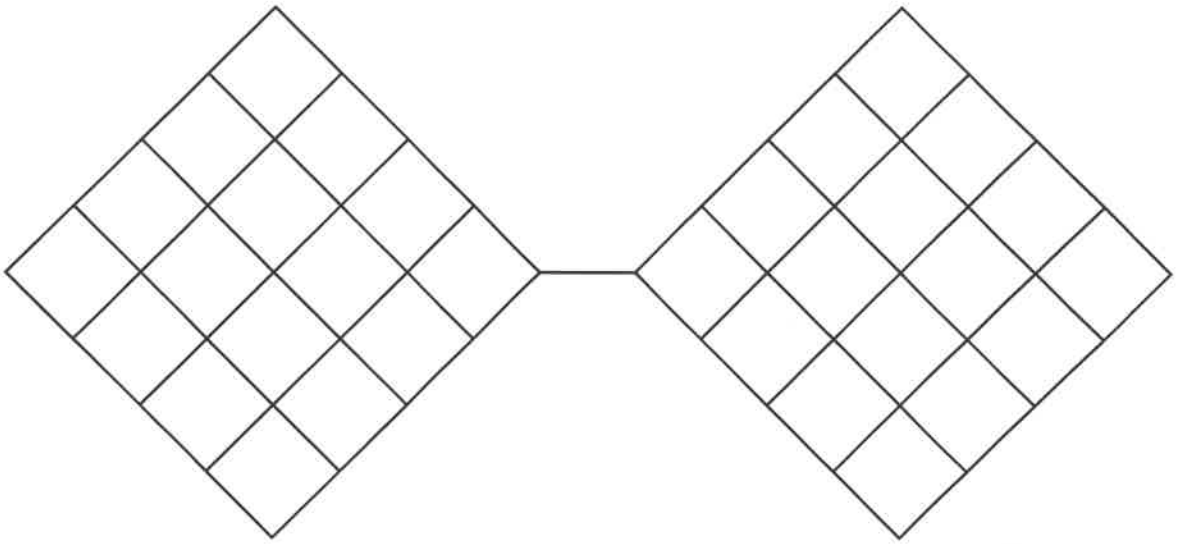


Figure 4.1: The Double Grid Graph

## 4.2 Further Work on Parallel Planar DAG Reachability Algorithms

There are also many opportunities to explore issues raised by my work on parallel planar DAG reachability.

The reduction algorithm is intended to be a general framework for developing parallel planar digraph algorithms. Thus it is reasonable to consider algorithms that it might be used to implement. Possible applications include the following:

- directed ear decomposition;
- topological ordering; and
- computing dominators in reducible flow graphs

Most of these algorithms have a connection to the general problem of reachability; this is not surprising given the original motivation for the reduction algorithm.

Application possibilities also extend to using algorithms built on the general reduction procedure into larger pieces of software. For example, one of the initial motivations for this work was a set of questions that arose from parallel compilation work. For structured languages without procedure calls, the reducible flow graphs of programs should be planar. An interesting question is whether a suitable set of restrictions on procedure calls can be formulated such that the flow graphs remain planar. Questions of what tasks (e.g., particular optimizations) can be implemented using the reduction procedure.

Implementation provides another area for exploration. The algorithm is quite complicated given the number of rules, the maintenance of connectivity pointers, and the need for cleanup. Can the algorithm be simplified in ways that make it easier to implement? Possible ways to simplify the algorithm are included in the following list:

- Reduce the number and the complexity of the rules needed to get an algorithm with provable performance on all inputs. This may be possible, though I have not yet had success in doing it.
- Drop certain rules, even though the bounds on running time may not be provable for all inputs. This approach is particularly interesting if one can demonstrate classes of inputs for which the running time is unchanged.
- Change other aspects of the algorithm such as cleanup. Is there some way to handle connectivity pointers in application-specific processing such that they no longer cause problems for the reduction rules? A danger in this approach is that it may just shift work (or complexity) from one place to another.

An area that I have investigated with success, but have not yet documented, is expanding the reduction algorithm to work with planar graphs containing cycles. This is particularly useful because it allows the development of algorithms for problems such as computing strongly connected components. With such an algorithm, my techniques would be sufficient for computing many-source reachability for any planar digraph: first compute strongly connected components, contract them, compute many-source reachability for the resulting DAG, then expand the DAG back out to the original graph, extending the reachability information to each strongly connected component.

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

- Two new rules (an arc contraction rule and an arc removal rule) for cycle faces are needed. The structural invariant changes to allow cycle faces as well as flow faces.
- In addition to crosspointers on flow faces, it is necessary to keep backpointers to the highest point reachable on the same side of the face.
- Cleanup is more complex because of the backpointers. One must now clean up two levels of arcs from sources or sinks. In addition, it takes $O(\log n)$ time to determine 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.

# Bibliography


[AG84] B. Aspvall and J. R. Gilbert. Graph coloring using eigenvalue decomposition. *SIAM Journal on Algebraic and Discrete Methods*, 5(4):526–538, December 1984.

[AGM87] N. Alon, Z. Galil, and V. D. Milman. Better expanders and superconcentrators. *Journal of Algorithms*, 8:337–347, 1987.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Alo86] N. Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[AM85] N. Alon and V. D. Milman. $\lambda_1$, isoperimetric inequalities for graphs, and superconcentrators. *Journal of Combinatorial Theory, Series B*, 38:73–88, 1985.

[Bar82] Earl R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM Journal on Algebraic and Discrete Methods*, 3(4):541–550, December 1982.

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

[Bop87] R. Boppana. Eigenvalues and graph bisection: An average-case analysis. In *28th Annual Symposium on Foundations of Computer Science*, pages 280–285, Los Angeles, October 1987. IEEE.

[CDS79] D. M. Cvetković, M. Doob, and H. Sachs. *Spectra of Graphs*. Academic Press, New York, 1979.

[CGT94] Tony F. Chan, John R. Gilbert, and Shang-Hua Teng. Geometric spectral partitioning. Technical Report CSL-94-15, Xerox PARC, July 1994. Revised January 1995.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. The MIT Press and Mcgraw-Hill Book Company, 1990.

[DH73] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17:420–425, 1973.

[Fie73] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305, 1973.

[Fie75] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(100):619–633, 1975.

[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.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. Freeman, San Francisco, 1979.

[GL89] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1989.

[GM92] Stephen Guattery and G.L. Miller. A contraction procedure for planar directed graphs. In *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 431–441, San Francisco, January 1992. ACM-SIAM.

[GM94] Stephen Guattery and G.L. Miller. On the performance of spectral graph partitioning methods. Technical Report CMU-CS-94-228, Carnegie Mellon University, December 1994.

[GM95a] Stephen Guattery and G.L. Miller. A contraction procedure for planar directed graphs. Technical Report CMU-CS-95-100, Carnegie Mellon University, May 1995.

[GM95b] Stephen Guattery and G.L. Miller. On the performance of spectral graph partitioning methods. In *6th ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, San Francisco, January 1995. ACM-SIAM.

[GMT95] John Gilbert, G.L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. In *9th International Parallel Processing Symposium*, Santa Barbara, April 1995. IEEE.

[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.

[HK92] Lars Hagen and Andrew B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEE Transactions on Computer-Aided Design*, 11(9):1074–1085, September 1992.

[Kao93] Ming-Yang Kao. Linear-processor NC algorithms for planar directed graphs I: Strongly connected components. *SIAM Journal on Computing*, 22(3):431–459, June 1993.

[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*, pages 181–192, 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.

[KS93] Ming-Yang Kao and Gregory E. Shannon. Linear-processor NC algorithms for planar directed graphs II: Directed spanning trees. *SIAM Journal on Computing*, 22(3):460–481, June 1993.

[LR88] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on the Foundations of Computer Science*, pages 422–431. IEEE Computer Society, October 1988.

[Lub86] M. Luby. A simple parallel algorithm fot the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, November 1986.

[Moh88] B. Mohar. The Laplacian spectrum of graphs. In *Sixth International Conference on the Theory and Applications of Graphs*, pages 871–898, 1988.

[Moh89] Bojan Mohar. Isoperimetric numbers of graphs. *Journal of Combinatorial Theory, Series B*, 47:274–291, 1989.

[MTV91] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *32nd Annual Symposium on Foundations of Computer Science*, pages 538–547, Puerto Rico, Oct 1991. IEEE.

[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.

[PSL90] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, July 1990.

[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.

[Sim91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[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.

[Wil65] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford, 1965.

[YG73] D. M. Young and R. T. Gregory. *A Survey of Numerical Mathematics*, volume II of *Addison-Wesley Series in Mathematics*. Addison-Wesley, Reading, MA, 1973.