A Parallel Algorithm for Finding a Separator in Planar Graphs *

Hillel Gazit Gary L. Miller

Department of Computer Science
University of Southern California
Los Angeles, California 90089

Abstract

We present a randomized parallel algorithm for finding a simple cycle separator in a planar graph. The size of the separator is $O(\sqrt{n})$ and it separates the graph so that the largest part contains at most $\frac{2}{3} \cdot n$ vertices. Our algorithm takes $T = O(\log^2(n))$ time and $P = O(n + f^{1+\epsilon})$ processors, where n is the number of vertices, f is the number of faces and ϵ is any positive constant. The algorithm is based on the solution of Lipton and Tarjan [8] for the sequential case which takes $O(n)$ time. Combining our algorithm with the Pan and Reif [12] algorithm, enables us to find a *BFS* of planar graph in time $O(\log^3(n))$ using $\frac{n^{1.5}}{\log(n)}$ processors. Using a variation of our algorithm we can construct a simple cycle separator of size $O(d \cdot \sqrt{f})$ were d is maximum face size.

1 Introduction

Many problems can be performed fast in parallel. Probably one of the easiest to explain is how to construct the transitive closer of a matrix using matrix multiplication and doubling-up. As a consequence of this, one can construct the *BFS* of a graph in $O(\log^2(n))$ using n^3 processors[1]. Since *BFS* of a graph can be performed in $O(n + m)$ time sequentially, where m is the number of edges, it is not clear that *BFS* as described above will ever be of practical value. The goal of this paper is to find a fast parallel algorithm which use a minimal number of processors. A related problem to the *BFS* of a graph problem is that of finding separators of a graph:

A subset of vertices B is a **separator** if the remaining vertices can be partitioned into 2 sets A and C such that there are no edges from A to C, and $|A|, |C| \leq \frac{2}{3} \cdot n$. The sets A, B, C form a partition of V.

A useful method for solving many problems is "divide-and-conquer". This method has been used to design sequential as well as parallel algorithms. In graph problems (and related problems), we usually need to find a separator in the graph in order to apply "divide and conquer".

Finding a small separator can be used to solve problems such as layouts for VLSI [6] [16], nested dissection [7] in numerical analysis, and for efficient message routing [2]. In particular Pan and Reif [12] showed how to solve the *BFS* problem given a family of small separators.

The simplest class of graphs with small separators is trees. Trees have separators of size 1. We will use this fact throughout this paper. Another important class of graphs with small separators is the planar graphs. A *planar graph* is a graph $G(V, E)$ that can be drawn on the plane, so that there are no edges crossing. Many problems in VLSI layout and Numerical Analysis are posed for planar graphs.

Lipton and Tarjan [8] showed that for every planar graph we can find a separator B of size $\sqrt{8 \cdot n}$. Note that their result is *optimal* up to multiplicative constants in the worst case, since there exist planar graphs for which the smallest separator is of size $O(\sqrt{n})$. The $\sqrt{n} \times \sqrt{n}$ grid is an example.

In a planar graph every **simple cycle separates** the graph to the "inside" and the "outside" of the cycle. A simple cycle separator is useful in "divide and conquer" when we unite partial solutions; and for communication problems [2]. The second author proved that in every planar graph we can find a simple cycle separator of size $\sqrt{8 \cdot d \cdot n}$ [10].

1.1 Previous Results

In 1979 Lipton and Tarjan [8] presented a sequential algorithm for finding a separator of size $\sqrt{8n}$ for planar graphs. Their results were improved in 1981 by Djidjev [1], to get a separator of size $\sqrt{6n}$. The first author [3] improved the size of the separator to $\frac{7}{3} \cdot \sqrt{n}$. In 1984 the second author showed that a simple cycle separator, as defined in [10], exists and can be found efficiently.

All these algorithms require a *BFS*[2] of the graph as input. In particular, given a *BFS* of a planar graph a simple cycle separator can be found in time $O(\log(n))$ using an

*This work is supported by National Science Foundation grant DCR-8514961

[1] The complexity of *BFS* of a directed graph was improved to the complexity of matrix multiplication over the integers [5].

[2] A *BFS* of a graph with respect to some vertex s is a *labeling* of the vertices such that the label of a vertex v is the distance from s to v.

0272-5428/87/0000/0238$01.00 © 1987 IEEE

optimal number of processors [10]. However the algorithm assumes that a planar embedding and a breadth first search of the graph are given. Thus, one has reduced the problem of finding a small separator to the *BFS* problem. But all known parallel implementations of *BFS* use matrix multiplication and thus a processor-time complexity, $P \cdot T$ of $O(n^3 \cdot \log(n))$. Although the complexity of the *BFS* was improved recently by us [5] to $P = M(n)$ processors[3] and $T = O(\log^2(n))$ time, the processor-time complexity is at least $O(n^2)$, which is the trivial lower bound for Matrix Multiplication.

In 1985 Pan and Reif [12] gave a reduction of *BFS* to the problem of finding a family of separators. They showed that given a family of $\sqrt{n}$ separators, they can compute the *BFS* of a graph in $T = O(\log^3(n))$ time and $P = O(\frac{n^{1.5}}{\log(n)})$ processors.

Recently we [4] showed that a small separator of a planar graph can be found in $\sqrt{n} \cdot \log(n)$ time using $\frac{\sqrt{n}}{\log(n)}$ processors, without requiring the *BFS* of the graph as input. Although this result gives optimal processor-time it is not a polylogarithmic time algorithm. But, because the constant in this algorithm are very small we recommend it for actual implementation.

In this paper we present a quite different algorithm for finding a separator. Our new algorithm uses $T = O(\log^2(n))$ time, and $P = O(n + f^{1+\epsilon})$ processors, where ϵ is any positive constant. Combining our algorithm with the Pan and Reif's algorithm [12], we can also compute *BFS* of a planar graph in $O(\log^3(n))$ time using $O(\frac{n^{1.5}}{\log n})$ processors. The same algorithm, with same complexity, can be used for solving single source shortest paths problem with arbitrary weights on the edges, as well as, solve linear systems which possess an underling planar graph, see [13].

1.2 Informal Description

In this section we give an informal description of our algorithm to find a simple cycle separator in a biconnected planar graph. In the following discussion n will denote the number of vertices, f the number of faces, and d the maximum face size. Recall that a 2-connected planar graph has a simple cycle separator of size $\sqrt{8d \cdot n}$[10]. Note that the number of vertices in a biconnected planar graph is at most $f \cdot d/2$, since every face has at most d vertices, and every vertex belongs to at least two faces. Using this fact we can rewrite the simple cycle separator theorem in terms of faces as $2 \cdot d \cdot \sqrt{f}$. [4] The new formula is the basis of our algorithm. We want to reduce the number of faces without increasing the maximal face size too much.

[3] In the rest of the paper $M(n)$ will denote the number of operations required to multiply matrices size n^2 over the ring of integers. The best known result is $M(n) = n^{2.36}$.

[4] We can prove that the separator size is $O(\sqrt{\sum_{F \in G}(\text{size of face F})^2})$, but we do not need this stronger result.

Our idea is to repeatedly union adjacent faces, until their number is small enough, so that we have enough processors to perform *BFS* using matrix multiplication. In this process, we have to take care that at most 2/3 of the original graph is contained in any face, that the size of the faces does not increase too much and the remaining graph is still 2-connected. When the number of faces is small enough, we find a *BFS* of the faces [10] and then use Miller's algorithm to find a simple cycle separator.

To keep track of what we have removed from the graph to form a face, we will assign weights to the faces as in [10]. The advantage is that when we union faces, the weight of the new face is the sum of the weights of the faces, vertices and edges inside the new face. Thus, our algorithm and Miller's algorithm differ from previous methods in that the weights are on faces, vertices and edges, rather than only on vertices, and the sum of all the weights is 1.

The main problem in our algorithm is how to union the faces in such a way that the new faces will not have long boundaries. The naive approach of choosing at random some adjacent face and uniting with it may yield a long boundary. Our idea is to union a "neighborhood" of faces. For that we need to find the neighbors of every face. We define neighbors by **number** and not by **distance**. That is, we want the k nearest neighbors of each face. Note that they can have a distance k in a long and narrow graph or 1 in a wagon-wheel graph. These *k-neighboring* faces are united together. We can perform *BFS* on their union and, as we later show can find some *BFS* layer which is small and close to the boundary. The method is similar to Lipton and Tarjan [8].

We pick a maximal disjoint subset of the k-neighborhoods as the bases of our construction. That is, we define a graph G^e in which vertices are the *k-neighborhoods*, and two vertices share an edge *iff* the *k-neighborhoods* they represent share a face. Finding a *maximal independent set* in this graph, enables us to find a small set of *k-neighborhoods*, such that every *k-neighborhood* is close in the number metric to some member of that set.

1.3 The Algorithm - A General Outline

Let $G(V, E)$ be an embedded planar graph, n the number of vertices, f the number of faces, d the maximal face size of any face, $P = F^{1+\epsilon}$ the number of processors for some positive constant ϵ and $k = n^{\epsilon/2}$. We will show that the *while* loop of the algorithm below will be executed a finite number of times.

The Algorithm

Let *Reduce-G*$(G(V, E), k)$ be an operation that reduces the planar graph $G(V, E)$ into another embedded planar graph $G'(V', E')$ such that G' is a subgraph of G and the number of faces in G' is $O(f/k)$. We describe this operation in detail later.

Our algorithm is:

1. $G_0(V_0, E_0) \leftarrow G(V, E); i \leftarrow 0;$
2. **while** $P < M(f_i -)$ **do**
   $i \leftarrow i + 1; k \leftarrow \lfloor \sqrt{\frac{P}{f_i}} \rfloor$
   $G_i(V_i, E_i) \leftarrow$ *Reduce-G*$(G_{i-1}(V_{i-1}, E_{i-1}), k);$
3. Find a face *BFS* of the faces G_i.
4. Find a simple cycle separator in G_i (using Miller's algorithm [10]).

2 Basic Parallel Algorithms

In this section we present several parallel algorithms which we will use later. The first finds the k nearest neighbors of each face. We will call these the **k-neighborhoods**. This algorithm also gives us a *BFS* in faces within each k-neighborhood of every face. The second algorithm finds a maximal set of k-neighborhoods such that each pair of k-neighborhoods are face disjoint. The third algorithm, using the results of the first and second algorithms, finds a face *BFS* of the graph from the boundaries of the independent set of k-neighborhoods. This gives us a set of larger disjoint neighborhoods of G. Between each layer of faces we also get a layer in G of edges. We divide each layer of edges into simple cycles and get a tree. We compute the weight inside and outside each cycle. We call the boundaries between faces that belongs to different of these larger neighborhoods the Voronoi Diagram of the graph. The fourth algorithm generates a Voronoi Diagram of the graph.

2.1 Finding k-neighbors

This section is based on Miller's [10] Breadth-First Search in planar graphs. We define a new graph $\hat{G}(\hat{V}, \hat{E})$ such that $\hat{V}$ is the set of the faces of G, and there is an edge between two faces *iff* they share a vertex in G. The distance between two faces is the length of the minimal path between them in $\hat{G}$. Note that $\hat{G}$ is not necessary planar.

To find the k nearest neighbors of each face, we want to find for every face F a set of k faces, such that the distance from F to every face outside its set is greater than or equal to the distance from F to every face in the set. Note that the k nearest neighbors are not unique. We call any such set the **k-neighborhood** of F, denoted by $C(F)$.

Choose k such that $P \geq f \cdot k^2$ where P is the number of processors and $k = f^{\epsilon/2}$. In this algorithm we assign k^2 processors per face, and for each face F we create $C(F)$ - a subset of faces of size k that form a sub-connected component, such that F is the *center*. Using the "doubling up" technique, we can complete this step in $O(\log(n))$ time.

Assume that for every face F we have:

1. $C(F)$ - a vector of length k, which represents the subcomponent of F.
2. $W(F)$ - a $k \times k$ matrix, used as a working space.

Each element of $C(F)$ and $W(F)$ is a pair ⟨face, distance⟩. Initially $C(F)$ contains only the immediate neighbors of F (distance 1).

Note that a face may have a large number of neighbors, so that naively collecting all its neighbors may take $O(n^2)$. The solution is that every vertex v, with degree more than two, computes a list of its neighbors (if there are more than k neighbors, then just use the first k). Using this information, every face can union the lists of all its neighbors, and drop duplicates using sort. The complexity is $O(\log(d \cdot k))$. (We assume that $k \geq d$. If the assumption is wrong, we can check first those faces that share an *edge*).

After every face has a list of its neighbors we iterate at most $\log(k)$ times to obtain a k-neighborhood. In each iteration, for every $F' \in C(F)$, we copy $C(F')$ into a distinct row $W(F)$ (updating the distances), then sort $W(F)$, remove duplicates so that for each face in $W(F)$ we keep only the shortest distance from F found so far. Then copy the first non-nil elements (up to k elements) back into $C(F)$.

procedure *Find-k-Neighbors*$(G(V, E), k)$
  for every face $F \in G$ **in parallel do**
    $C(F) \leftarrow \{\langle F', 1\rangle | F'$ and F share a vertex$\}$;
  for $i := 1$ **to** $\lceil \log(k) \rceil$ **do**
    for every face $F \in G$, $1 \leq i, j \leq k$ **in parallel do**
      Copy the neighbors of every face in $C(F)$
      into $W(F)$. Compute their distance from F.
      SORT(W(F)) to drop duplicates.
      SORT(W(F)) by distance.
      Copy the first (up to k) elements of $W(F)$ to $C(F)$.
end *Find-k-Neighbors*

Definitions

1. Let $C^i(F)$ denote $C(F)$ before the i^{th} iteration.
2. Let *distance*(F, F') be the minimal distance (in faces) between F and F'.

Lemma 2.1 $\forall i, \forall F \in G$,
$|C^i(F)| = \min(k, |\{F' | distance(F, F') \leq 2^i\}|)$

Proof: The proof will be given in the full version of the paper. □

Lemma 2.2 *Assuming that the graph is connected and $k \leq f$, after the algorithm Find-k-Neighbors every face F knows all the distances to its k nearest neighbors (which belong to $C(F)$).*

Proof: Since the graph is connected and $k \leq f$, there are at least k faces a distance k or less from F. After $\lceil log(k) \rceil$ iterations $C(F)$ contains k vertices by Lemma 2.1. □

Lemma 2.3 *The algorithm takes $T = \log^2(n)$ time and $n + f \cdot k^2$ space using $P = MAX(n, f \cdot k^2)$ processors.*

Proof: Using k^2 processors we can set the elements of $W(F)$ in one step, and sort k^2 in $O(\log(k))$ time (and quite efficiently, since we can use integer sorting). Removing duplicate elements can be done in constant time, compressing $W(F)$ (to remove null elements) in $O(\log(k))$ time, and copying the first non-nil (upto k) elements into $C(F)$ can be made in constant time. We need n processors and $O(\log(n))$ time for the distance 1 list. We use $O(k^2)$ space per vertex, so $O(n \cdot k^2)$ space total. □

2.2 Finding an Independent Set

In this subsection we find a maximal set of k-neighborhoods, *MIS*, such that each pair of neighborhoods contain a disjoint set of faces. Recall that each k-neighborhood is a set of k faces. Thus, the number of neighborhoods in *MIS* is at most f/k. The idea is to define a new graph $G^c(V^c, E^c)$ such that V^c is the set of faces of G, and in which two faces F, F' share an edge in E^c *iff* their neighborhoods $C(F), C(F')$ share a face. *MIS* is an independent set in this graph.

Let $G(V, E)$ be a planar graph such that every $F \in G$ has a vector $C(F)$ of length k containing the k nearest vertices and the distances from F to them. $C(F)$ was computed by the *Find-k-Neighbors* algorithm described in the previous section.

Note that G^c is an *intersection graph*. That is, a graph in which the vertices are sets of elements, and there is an edge between two sets *iff* they have a common element. In our case, the sets are the k-neighborhood of each face. Our goal is to find a maximal independent set for this graph.

Luby [9] has presented an algorithm to find a maximal independent set in $O(\log(n))$ time with $O(m+n)$ processors, where m is the number of edges. However, G^c is not necessarily a planar graph, and may have a very large number of edges. For example, consider a *star*: For $k \geq 2$ the center face will belong to all subcomponents, so G^c is actually a clique. Since we have only $P = n \cdot k^2$ processors, we may not have enough processors. We will give a general algorithm for finding a maximal independent set in an intersection graph, where P is equal to the input size (which is the sets description), and T is $O(\log(n) \cdot \log(\log(n)))$ for the *CRCW* model and $O(\log^2(n))$ for the *EREW* model. The same algorithm can be used for other intersection graph problems such as maximal matching, with the same processor and time complexity which were mentioned above.

The following algorithm is Luby's independent set [9] algorithm.

procedure *Luby-Independent-Set*$(G(V, E))$
  MIS$\leftarrow \emptyset$;
  while $V \neq \emptyset$ **do**
    Each vertex picks a random number of size $(1, n^4)$
    for all $v \in V$ **in parallel do**
      if v has a number greater then all its neighbors
        then add v to *MIS*;
          remove v and its neighbors from V; **fi**
      fi
    od
  od
end *Luby-Independent-Set*

Luby proved that with high probability this algorithm will halt in $O(\log(n))$ iterations.

We wish to compute a maximal independent set for $G^c(V^c, E^c)$, but since we do not have enough processors to do it directly, we instead compute for every face F the list $L(F)$ of all F' such that $F \in C(F')$. We show that finding a maximum of the lists of all vertices in F's subcomponent is equivalent to finding the maximum in the set of neighbors of F in G^c. We will call this set *MIS*.

The procedure *Find-Containing-Components* finds, for every F, the list $L(F) = \{F' | F \in C(F')\}$.

procedure *Find-Containing-Components*$(G(V, E))$
  Create an array LL of length $f \cdot k$;
  for every face $i \in G$, $0 \leq j < k$ **in parallel do**
    $LL[k \cdot i + j] \leftarrow$ the j^{th} neighbor of face i.
  od
  Sort LL in lexicographic order;
  for every $i \in V$ **in parallel do**
    use binary-search on LL to find where $L(i)$ begins;
    use binary-search on LL to find where $L(i)$ ends;
  od
end *Find-Containing-Components*

Lemma 2.4 *Procedure Find-Containing-Components takes $O(\log(f))$ time with $O(f \cdot k)$ processors.*

Proof: Since for every face F, $|C(F)| = k$, we can set the array LL in constant time using $f \cdot k$ processors. Both the SORT and the SEARCH on an array of length $f \cdot k$ takes $O(\log(f \cdot k)) = O(\log(n))$, by the definition of k.

If we try to run Luby's algorithm we have to solve two problems:

1. How to check if a face is greater than all it neighbors.
2. How a face can notify its neighbors that it is in the independent set.

We will use the L's we computed in the previous algorithm as "communication centers". Every set of processors that was assigned to some list L will check which is the maximum and if it is unique. After the maximum is found, all the processors of the smaller elements of the set notify their faces that they are not in the set. If a face does not get such a message, this face is in the set.

The second problem is solved in a similar way. If a face is in the independent set, all its k neighbors notify their L set to delete the face they belong to from the graph.

Lemma 2.5 *The complexity of the algorithm in the CRCW model is $O(\log(n) \cdot \log(\log(n)))$ with $O(n \cdot k)$ processors.*

Proof: In his paper [9] Luby proves that with high probability $O(\log(n))$ iterations are suffice. Each iteration can be implemented in $O(\log(\log(k))) \leq O(\log(\log(n)))$ time, since the max operation requires $O(\log(\log(k)))$ time in *CRCW* model [15], and all other operations take constant time in the *CRCW* model. □

Lemma 2.6 *The algorithm applied on $G(V, E)$, computes a maximal independent set for the graph $G^c(V^c, E^c)$.*

Proof: The proof will be given in the full version of the paper. □

Lemma 2.7 *MIS contains at most f/k k-neighborhoods*

Definition 2.8 *An I-set is any region obtained from a k-neighborhood in MIS by removing the last layer if it is not a complete layer of $\hat{G}$.*

Since each I-set consists of full layers in the *BFS* in $\hat{G}$ their boundary can be written as nonnesting simple cycles, see [10].

2.3 Computing planar graph layering

In this section we compute the distance of each face not in an I-set to the nearest I-set boundary. We will then use this information to find a noncrossing *BFS* spanning forest of the faces not in any I-set. We can then combine this *BFS* with each I-set *BFS* spanning tree obtaining a *BFS* spanning forest for all of $\hat{G}$. We start by constructing for each face not in an I-set the distanceto the boundary of an I-set. We present the algorithm in procedure form, see Figure 1.

Procedure: *Find-layers*

1. Mark every face in an I-set level 0.
2. Mark every face level 1 that shares a vertex with an I-set boundary and is not marked.
3. In parallel for each face F not marked compute the distance $d(F)$ to the nearest face in its k-neighborhood which is marked 1.
4. For all unmarked faces mark them $1 + d(F)$.

Figure 1: Finding the Distance to Your Nearest I-set.

To see that the algorithm marks all face we must show that every k-neighborhood contains a face marked 1. We state this as a lemma:

Lemma 2.9 *Every k-neighborhood which is not in I-sets contains a face marked 1.*

Proof: Since *MIS* is a maximal independent set all k-neighborhoods must contain a face from some set in *MIS*. It will suffice to see that for each set S in *MIS* all faces are marked 0 or 1. But every face in S is either in the I-set of S or sharing a vertex with it. □

Lemma 2.10 *The Procedure Find-layers uses $O(\log(k))$ time and $n + f \cdot k$ processors.*

Proof: We can check if the last layer is full or not in $O(\log(k))$. Finding level 1 can be made in time $O(1)$ by every vertex checking if any of its faces are in level 0; and then every face checks its vertices if any touch a face in level 0. Every face can check its component in $\log(\log(k))$ time [15]. □

Using the layering of the faces that we have just computed we want a *BFS* spanning forest from the boundaries of the I-sets. The spanning forest is in the face-incidence graph $\hat{G}$. But $\hat{G}$ need not be planar. We will construct the forest such that it is noncrossing:

Definition 2.11 *We say a subgraph H of the face-incidence graph $\hat{G}$ of G is* **noncrossing** *if no two edges of H cross, that is, there do not exist four faces A, B, C, D of G that share a vertex x in the order (A, B, C, D) and A is connected to C in H via x and B is connected to D in H via x.*

We say that T is a **BFS (spanning) forest from a subset** S of the vertices of G if T is a (spanning) forest and every component of T contains exactly one vertex from S and the paths to this vertex are shortest paths to S.

Procedure *BFS-Forest*

1. **If** a face F of mark $i+1$ shares an edge with a face of mark i **then** pick such face F' and set the parent of F to F'.
   Else pick the face F' with mark i and smallest index which shares a vertex with F and set the parent of F to F'.

Lemma 2.12 *Procedure BFS-tree generates a noncrossing BFS spanning forest of $\hat{G}$ from I-set boundaries.*

Later we will break every I-sets component into layers by their distance from the center. This is easy to do because we have the *BFS* tree of every component.

2.4 Computing Induce Weights on a Set of Cycles

In this section we consider the problem of given G and a collection of simple cycle $C_1, \ldots, C_t$ find the interior and exterior weight of each cycles. We could always solve the problem for each cycle separately but this would give us an algorithm which uses $O(t \cdot n)$ processors. We will describe an algorithm which uses only $O(n + \sum_{i=1}^{t} size(C_i))$ processors for the case when no two cycle interlace.

Definition 2.13 *Two simple cycles A and B* **interlace** *in a planar embedded graph G if there exist two edge of B one in the interior of A and the other in the exterior.*

If the cycles $C_1, \ldots, C_t$ are vertex disjoint then no pair of cycles interlace. We get a natural tree defined by the regions of G with respect to these cycles.

Definition 2.14 *The* **regions** *formed by the cycles $C_1, \ldots, C_t$ are the equivalence classes of faces defined by the relation:*

$$F \equiv F' \text{ if no cycle } C_i \text{ separates } F \text{ from } F'$$

The definition is not trivial when the cycles share edges because regions may not be connected, see Figure 2. In the figure the "inside" and the "outside" are the same region.

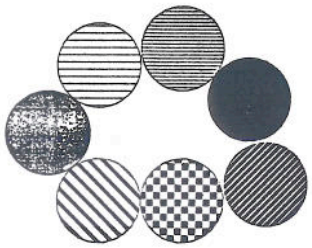


Figure 2: An Example of a Disconnected Region.

The **region-cycle tree** has a vertex for every region formed by the cycles and an edge for each cycle. A region vertex is adjacent to a cycle edge in the tree if the cycle shares an edge with the region. It follows that each region forms a connected subgraph in the geometric dual. Using this fact we can compute each region and its weight. The weight of a region does not include the weight on it boundary. The weight of a tree vertex is the weight of its corresponding region and the weight of a tree edge is the weight of its corresponding cycle. Thus we can determine the tree and its weights. We can then use either the Tree-Tour [14] or the Parallel Tree Contraction [11] to find the weight on the interior of each cycle and thus the exterior weights, see [10].

If the cycles $C_1, \ldots, C_t$ are not disjoint then the problem is slightly more complicated. In this case the regions may no longer form connected components in the geometric dual G^*. If we want the regions and cycles to form a tree we must even introduce empty regions. Finally the weight of cycle vertices in the region-cycle tree is not well defined since several cycles may share vertices and edges. In all the applications in this paper the unweighted region-cycle tree is either given or easy to construct. We have found an algorithm to find this tree which we will present in the full paper. Assume for this paper that the unweighted region-tree is always given.

Our idea is to use Parallel Tree Contraction to both find the regions and compute the induced weights on the cycles simultaneously. In $O(\log n)$ time we can remove all edges of G which do not belong to any of the cycles. Notice now that the leaves of the region-cycle tree are simply those faces whose boundary is one on the cycles C_i. Thus the RAKE operation is straightforward. If a face F is a region and its boundary is a cycle C_i then we remove C_i from the list of cycles remove F and C_i from the tree, and remove the mark that indicates that edges of C_i belong to the cycle C_i. The COMPRESS operation is slightly messier. Here each cycle determines if it is of degree 2 in the region-cycle tree by inspection of the tree. Using the deterministic Parallel Tree Contraction algorithm, a constant fraction of these cycles which are independent in the tree remove themselves. Between phase of the Parallel Tree Contraction we remove all edge which do not belong to any cycle. During the expansion phase the induced weights on the cycles which were remove by COMPRESS can be evaluated.

Lemma 2.15 *Given an embedded planar graph G and a set of noninterlacing cycles $C_1, \ldots, C_t$ one can compute the induced weight on the exterior and interior of each cycle in $O(\log^2 n)$ time using at most $n + \sum_{i=1}^{t} size(C_i)$ processors.*

It it well known that every tree has either an edge which is a separator or a vertex whose removal decomposes the tree into subtrees of size at most $1/3$ of the weight of the original graph. In our application we view a planar graph as a tree where the vertices correspond to regions and the edges correspond to simple cycles. Since the cycles may share vertices and edges we cannot view our tree as simply weighted.

Lemma 2.16 *If G is a weighted 2-connected embedded planar graph and $C_1, \ldots, C_t$ is a collection of noninterlacing simple cycles, then either one of these cycles is a separator of G or one of the regions formed by the cycles has all its induced face weights at most $1/3$.*

Proof: The Lemma follows from the two facts: (1) the region-cycle graph is a tree and (2) the interior weight plus the exterior weights of any simple cycle is at most one. □

2.5 Voronoi Diagrams

Using Procedure *BFS*-Forest we can find a spanning forest for $\hat{G}$. In this subsection we define and compute the boundary between the trees in this forest as a subgraph of G. This subgraph has a strong analogy between it and the Voronoi diagram for geometric objects in the planar. We shall give a graph theoretical definition of Voronoi Diagrams.

Definition 2.17 *We say that a subgraph $H = (V', E')$ of G is the* **Voronoi Diagram** *of G with respect to a noncrossing (spanning) forest T of the face-incidence graph $\hat{G}$ if the edges E' of H are those edges e such that the two faces common to e belong to different components of T. The vertices V' are the vertices induced by E'. If T does not span $\hat{G}$ then we add to E' those edges such one of its faces belong to T and the other*

does not. Thus H contains boundary edges between faces in T and those not in T. These edges can be decomposed into a set of nonnesting simple cycles which we call **boundary cycles.**

To continue the analogy between Voronoi Diagrams for geometric objects and topological objects, let S be a set of faces of G such that every face is contained in exactly one tree of T. We call H the **Voronoi Diagram** of S with respect to T. This definition can be extended from a set of faces to any set of nonnesting cycles where T is a forest created by from the cycles. We call these the **base cycles** of the (spanning) forest as well as the base cycles of the Voronoi diagram. Note that we often pick T to be a *BFS* forest from the base cycles.

In the case of a single source *BFS* searches of $\hat{G}$, the boundary between layers were always very nice. They decomposed into nonnesting simple cycles in a very natural way. Here we must be a little more careful. We do not want the decomposition into cycles to cause cycles to cross an edge of the forest T. Thus we want the boundary of a region to be decomposed into cycles as long as the cycles do not partition a region. We will simply call this "the boundary of the region **decomposed into cycles**". We discuss this further in the full paper. A planar graph G (in solid lines) and spanning forest (in thin lines) are shown in Figure 3. The boundary of the larger tree decomposed into cycles creates the two cycles: (a, b, c, d) and (e, f, g, h, i, j, k, l). By making two virtual copies of the vertex v, one common to the edges f, g and the other common to j, k, all the vertices of these two cycles are of degree 2, so we can think of both cycles as simple. We call these the **virtual vertices** of the Voronoi Diagram.

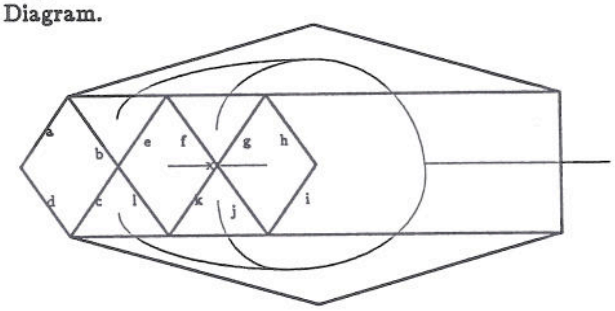


Figure 3: An Example of the Boundary of a Region Decomposed into Cycles.

Lemma 2.18 *The virtual Voronoi Diagram described above is a planar graph and the boundary of every face is a set of nonnesting simple cycles .*

Proof: The proof will be given in the full version of the paper. □

2.6 Creating a spanning tree

In this algorithm we want to take the faces *BFS* tree of faces and create a spanning tree of *vertices*. The distance between vertices of the same face is at most $d/2$. Therefore we expect the radius of the spanning tree of vertices to be at most the radius of the *BFS* tree of faces multiply by $d/2$, see figure below.

Definition 2.19 *Level(F) is the level of face F in the faces-BFS tree.*

Definition 2.20 *A* linkage vertex *is a vertex that belongs to two faces in different levels.*

procedure *Build-spanning-tree*$(G(V, E))$,
  for every face F in parallel do
    Give every vertex in F the value $\frac{d}{2} \cdot Level(F)$
    plus the distance to the nearest linkage vertex.
  od
  for every vertex v in parallel do
    Give v the minimum of values it got in the
    previous step. Link every vertex to its neighbor
    with the lowest value.
  od
end *Build-spanning-tree*

Lemma 2.21 *For every vertex v such that* $\text{dist}(v) > 0$, *there is some vertex u,* $\text{dist}(v) > \text{dist}(u)$.

Proof: For some face F, $dist(v) = \text{DIST}(u, F) = \frac{d}{2} \cdot \text{Level}(F) + distance(v, u)$ where u is a linkage vertex. Then there is a neighbor $w \in F$ such that $distance(v, u) > distance(v, w)$, so $dist(v) > dist(w)$. □

Lemma 2.22 *The complexity of the algorithm is* $O(\log(n))$ *time using n processors.*

Proof: The algorithm uses doubling up on the vertices around every vertex and every face, so the processors complexity is bound by the number of edges. □

3 Reduction Step

In this section we present our algorithms that takes a planar graph and returns a subgraph such that it still contains a small simple cycle separator. This separator will be a separator for the original graph. In particular we show:

Theorem 3.1 *There exist a parallel algorithm which takes an embedded weighted 2-connected, planar graph G with f faces, each face of size at most d, and no face weight $> 2/3$ and returns a subgraph H that is 2-connected with at most $3f/k$ faces, each of size at most $5 \cdot d \cdot \sqrt{k}$, and no induced face weight $> 2/3$. This algorithm will use $O(\log^2 n)$ time and $n \cdot k^2$ processors.*

The algorithm in Theorem 3.1 is the same as the operation Reduce-G described in section 1.3. We will call this algorithm *Reduce-G*.

We will assume that the graphs G and H have the following compact representation: Each chain - a maximal simple path with internal vertices of degree two, has been replaced by an edge plus a number indicating the length of the original chain. Onward this section let $G = (V, E)$ be an embedded weighted 2-connected planar graph with f faces, each of size at most d, and no face has weight $> 2/3$.

3.1 Finding a base cycle for each I-set

In Section 2 we found the k-neighborhoods of each face of G and a maximal independent set of these neighborhoods. In this subsection we reduce the general planar separator problem to the case where all I-sets have spanning trees of diameter $O(d\sqrt{k})$ we allow the size of the face which is the center of the *BFS* to be of size $O(d\sqrt{k})$. We call this cycle the **base cycle** of the I-set. Note that after running procedure Find-Base-Cycle the boundary of an I-set need not be a simple cycle. We will use the fact that each I-set has a base cycle in Subsections 3.2. In Subsection 3.3 we address the issue of the boundaries not being simple. We next give the main procedure, Find-Base-Cycle, in Figure 4.

If procedure Find-Base-Cycle has not found a separator then it has constructed a subgraph R of G. Note that the original *BFS* of $\hat{S}$ and its layering restricted to R is now a *BFS* and layering of $\hat{R}$ from the face C. This fact follows since C is in the old layering of S. Note also that the number of levels in $\hat{R}$ is at most $\sqrt{k}$. One must show that Miller's algorithm in step 7 will return a separator of size $O(d\sqrt{k})$.

Lemma 3.2 *We can find a layer inside every I-set such that the layer size plus twice the distance in (faces between) cycle and boundary plus is at most $2 \cdot \sqrt{k}$.*

Proof: The proof is similar to Lipton and Tarjan [8]. We give every layer a value the consist of it size plus twice the distance to the boundary and find a minimum. □

3.2 Finding a small diameter spanning forest

In the last subsection we show how to reduce the general case to the case where the boundary of each I-set is a simple cycle. In this section the goal is to find a spanning forest and

procedure *Find-Base-Cycle*
IN Parallel for each I-set S do

1. Compute the layers of the neighborhood S up to the last complete layer.
2. Divide each layer into simple nonnesting edge disjoint simple cycles.
3. Compute the interior and exterior weight of each simple cycle. If any cycle is a separator return this cycle, we are done.
4. Find the size of each layer. Compute for every layer a value, it size plus twice the distance from the last layer.
5. Find the layer L with the lowest value.
6. Use this layer to separate G into regions.
7. Pick a region R such that the induced weight on each of its faces is at most $1/3$.
8. If the region R contains the center face of the I-set S then apply Miller's algorithm to R using the *BFS* from S.
   Else Let C be the only simple cycle of L common to to R. Return R with base cycle C, and the *BFS* tree of $\hat{S}$ restricted to $\hat{R}$.

Figure 4: Finding a Base Cycle for each I-set.

its Voronoi diagram of these cycles satisfying the following four conditions:

1. The size of each base cycle or boundary cycle is $O(d\sqrt{k})$.
2. The induced weight on a base cycle is at most $1/3$.
3. The number in base cycles of the diagram is at most f/k.
4. The radius of the spanning forest is $O(d \cdot \sqrt{k})$.

Note that to maintain these conditions the base cycles will not necessarily be the boundaries of the I-sets. We give the algorithm in procedure form, Figure 5.

Lemma 3.3 *If a face F is in distance $i \geq \sqrt{k}$ from the nearest I-set boundary then there is a small cycle (size $d \cdot \sqrt{k}$) between it and the boundary.*

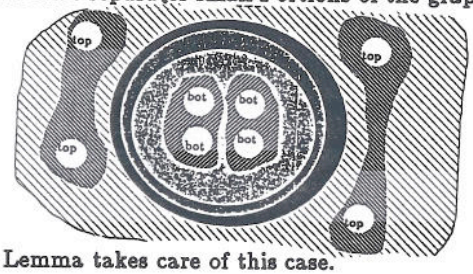
Proof: Assume not. Between every two layers we have at least $2 \cdot \sqrt{k}$ faces (to "give" enough edges to two layers with $d\sqrt{k}$ edges each). Therefore we have cycles of $2 \cdot \sqrt{k}$ faces each. k faces in these cycles are in distance $\sqrt{k}$ from every face in the top; (By going down to a layer, and then $\sqrt{k}/2$ to each side). That proves that the *k-neighborhood* of this face does not contain an independent set face, in contradiction to the way we built the I-sets. □
The previous Lemma proves that we can always find a small

procedure *Generate-the-Voronoi-diagram*

1. Using the distances computed by procedure Distance-to-I-set compute the layers of the *BFS* and decompose each layer into noninterlacing edge disjoint simple cycles.
2. Find all the simple cycles above of size at most $d\sqrt{k}$. Add to this set of cycles the base cycle each I-set computed by procedure Find-Base-Cycle.
3. For each simple cycle compute its interior and exterior weight.
4. If one of the cycles is a separator **then** return it and quit
   else Find a region R with all induced face weights at most $1/3$.
5. Using the natural partial order on the simple cycles in R given by the *BFS* forest, set the base cycles for R equal to the minimal cycles, (from step 2) of this order.
6. Construct the spanning forest T from the base cycles using procedure *BFS*-Forest. Use this forest to compute Voronoi diagram.

Figure 5: Finding Voronoi Diagrams

cut that separates the graph. near an i-set boundary If the heavy part is outside the cycle, we will use this cycle later to create new faces size $O(d \cdot \sqrt{k})$. If most of the weight is in the outside we know (from previous chapter) that the radius of the spanning tree of the I-set is at most $\sqrt{k}$. If most of the weight is inside the cycle we will go into the cycle, and drop the outside. The figure below demonstrates this case. The top cycles are near the I-sets, and the bot cycles are small cycles that separates small Portions of the graph. The



following Lemma takes care of this case.

Lemma 3.4 *Every cycle is in distance, in both directions*[5]*, of at most $\sqrt{k}$ from the nearest boundary or a small cycle.*

Proof: The proof is similar to the previous Lemma, and we will leave it to the reader. □
That means that when we go inside we still have small cycles inside near to each other.

[5] "inside" the cycle and "outside" the cycle

Lemma 3.5 *The BFS tree of $\hat{R}$ return in step 7 has radius of at most $2 \cdot d \cdot \sqrt{k}$.*

Proof: The radius of the tree inside the I-set is at most $\sqrt{k}$ (see previous section). And the distance from the boundary to the small cycle outside is at most $1 + \sqrt{k}$. □

3.3 Mesa partition

After applying algorithm Generating the Voronoi Diagram on the graph G, we have found at most $O(f/k)$ base cycles, discarded their interiors and found a Voronoi Diagram for them from a spanning forest of the base cycles in $\hat{G}$ of diameter at most $O(\sqrt{k})$. Each region of the Voronoi diagram contains exactly one base cycle but the boundary of the region need not even be a connected subgraph of G. In this subsection we show how to reduce the problem to the case when all the boundaries are (virtually) simple and form a 2-connected subgraph of G. Let R be one of the regions, C its base cycle, and $B_1, \ldots, B_t$ the boundary of R decomposed into cycles described in previous section.

Even though it is not necessary we give our intuitive picture of R. We think of R as a relief map with boundary C. The contour lines of this map are the layers. The map is of the Southwest region which has mesas. In R the mesa are the faces $B_1, \ldots, B_t$. Between these mesas are ridges. We show that in our case the ridges form a spanning tree of the mesas. Our map contains no local minima. We partition the mesas via the ridges. The formal definitions are given below.

Let T be the *BFS* spanning of $\hat{R}$ from C. We say that an edge (vertex) e (v) of R is a **ridge** edge (vertex) if the induced cycle in T from the two faces common to e (v) forms a nontrivial partition of the set $B_1, \ldots, B_t$ for $t > 1$. We will show that the ridge edges and vertices form a forest that spans the boundary cycles $B_1, \ldots, B_t$.

procedure *Mesa-Partion*

1. Find all ridge edges and vertices.
2. For every ridge vertex and face common to it find the path back to C.
3. For each path in $\hat{R}$ above find the induced path in R.
4. Viewing pairs of paths as noninterlacing simple cycle of R, compute the weight on the interior and exterior of each cycle.
5. If one of the cycles is a separator **then** return the cycle, we are done.
   else Find a region with all induced face weights $> 1/3$.
6. Return this region.

Figure 6: Determining the Simple Cycle Boundary for each I-set and Removing all Others

3.4 Computing the Face-Vertex Dual Subgraph

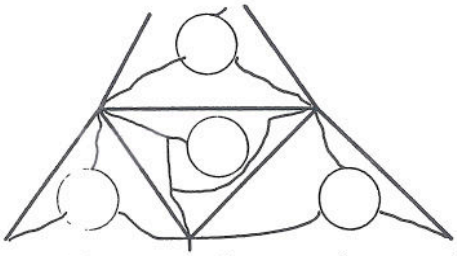
In this subsection we construct the graph as required in Theorem 3.1. From our input graph G we have constructed a possible subgraph of G which satisfies all the input conditions. Since a separator for this subgraph will be a separator for G we set G to be this new graph. By Subsection 3.2 we have found a Voronoi subgraph of G satisfying the four conditions.

Let us look on the Voronoi diagram. The vertices are sometimes common to faces of two components (type 1 vertices), and sometime to faces of three or more components. (type 2 vertices) There are paths between two type 2 vertices composed only of type 1 vertices. We will call these paths *meta-edges.*

Lemma 3.6 *The number of meta-edges is at most* $3 \cdot f/k$.

Proof: As we have shown in section 2, the extensions of I-sets create a planar graph. The *meta-edges* are the edges of this graph, and the Lemma follows immediately from Euler formula. □

The following Figures describe the cycles we expect to get.



Lemma 3.7 *We have a new face around every meta-edge. The face size is at most* $d \cdot (4 + 5 \cdot \sqrt{k})$

Proof: The *meta-edge* separates extensions of two I-sets. We create the face by paths from the centers of both of them to both sides of the *meta-edge*, and we may have a small cycle in both sides of the *meta-edge*. The total distance in faces is: $2 + 2 \cdot \sqrt{k}$ in each component, plus $(1 + \sqrt{k})$ for every path from each I-set to the small cycles, and we have four such paths. We can translate every face size d into a path of size $d/2$, so we get $4 \cdot d$ plus $4 \cdot d \cdot \sqrt{k}$. Inside the two small cycles we can go around the cycle using the shortest path, and get $d \cdot \sqrt{k}$ for both cycles together. By summing everything up we get the result we claim.

4 Applications of Reduce-G

4.1 Stopping the recursion

We apply *Reduce-G* repeatedly till one of the following conditions hold:

1. The procedure *Reduce-G* returns a subgraph H which is a simple cycle separator.
2. The number of faces is small enough to compute a *BFS* of the graph H via matrix multiplication in $O(\log^2 n)$ using P processors.

4.2 Number of iterations

If we apply the procedure *Reduce-G* repeatedly, we can pick a larger and larger value for k. Therefore we will need only $O(\log(\log(f)))$ iterations to find a separator using $O(n)$ processors. We show that for $P = n^{1+\epsilon}$ only a constant number of iteration is necessary.

Define f_i to be the number of faces in the reduced graph after iteration i, and k_i to be the k used in iteration i.

In the first iteration $P = f \cdot k^2$ and $f_1 = 2 \cdot \frac{f}{k}$, but in the second iteration $P = f_1 \cdot {k_1}^2$, so

$$P = f \cdot k^2 = (\frac{3 \cdot f}{k}) \cdot k_1^2 \Longrightarrow k_1 = k^{\frac{3}{2}}/\sqrt{3}$$

In the same way we can show that in the third iteration $k_2 = (k^{\frac{3}{2}})^{\frac{3}{2}}/3 = k^{((\frac{3}{2})^2)}/3$, and in the i^{th} iteration $k_i = k^{((\frac{3}{2})^i)}/3^{\frac{i}{2}}$. An upper bound for i can be obtained when $k_i \geq f$, since then we finish the algorithm. This implies $k_i = k^{(\frac{3}{2})^i}/3^{\frac{i}{2}} < f$.

If we take $P = 2 \cdot f$ ($k^2 = 2$), then an upper limit to the number of iterations is: $O(\log(\log(f)))$. If on the other hand we take $k = f^{\epsilon/2}$ where ϵ is some constant. then the number of iteration is a fixed constant i satisfying the equation $k^{\frac{3}{2}^i}/3^{\frac{i}{2}} = f$. Note that i is constant which only depends on $\frac{1}{\epsilon}$ for a large f. A small ϵ means a large constant, and vice versa.

The separator size is increased also with the number of iterations. Using the result of the previous section we can conclude that the separator size will be about $(3 \cdot 5^2)^{i/2} \cdot 2 \cdot d\sqrt{f}$

It is instructive to compute the number of iterations for a few values of P. If the number of processors $P = n^{1.5}$ and regular n^3 matrix multiplication is used, then two iterations will suffice. For $P = n^{1.8}$ we need only one iteration. Using the processor efficient matrix multiplication these exponents can be improved.

5 Conclusion

As mentioned in the Introduction we have found an optimal algorithm for the separator problem that needs $O(\log(n) \cdot \sqrt{(n)})$ time, see [4]. It is open whether an optimal polylogarithmic algorithm exists.

Another open question is the *BFS* of planar graphs. Pan and Reif [12] presented a good algorithm for *BFS*, but the product $P \cdot T$ is about $n^{1.5}$. Can we find a *BFS* of a planar graph in a more efficient way?

References

[1] H.N. Djidjev. On the problem of partition planar graphs. *SIAM J. Alg. Disrete Math.*, 3(2):229–240, June 1982.

[2] Greg N. Fredrickson and Ravi Janardan. Separator-based strategies for efficient message routing. In *27st Annual Symposium on Foundation of Computer Science*, pages 428–437, IEEE, Oct 1986.

[3] H. Gazit. An improved algorithm for separating a planar graph. manuscript.

[4] H. Gazit and Gary L. Miller. An $O(\sqrt{n}\log(n))$ optimal parallel algorithm for a separator for planar graphs. manuscript.

[5] H. Gazit and Gary L. Miller. A parallel algorithm for bfs of a directed graph. manuscript.

[6] C. Leiserson. Area-efficient graph layouts (for vlsi). In *21st Annual Symposium on Foundation of Computer Science*, pages 270–281, IEEE, Oct 1980.

[7] R.J. Lipton, D.J. Rose, and R.E. Tarjan. Generalized nested dissection. *SIAM J. on Numerical Analysis*, 16:346–358, 1979.

[8] R.J. Lipton and R.E. Tarjan. A seperator theorem for planar graphs. *SIAM J. of Appl. Math.*, 36:177–189, April 1979.

[9] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, November 1986.

[10] G.L. Miller. Finding small simple cycle seperator for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, June 1986.

[11] G.L. Miller and J.H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, IEEE, Portland, Oregon, 1985.

[12] V. Pan and J. H. Reif. *Extension of Parallel Nested Dissection Algorithm to the Path Algebra Problems.* Computer Science Department TR-85-9, State University of New York at Albany, 1985.

[13] Victor Pan and John Reif. Efficient parallel solution of linear systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 143–152, ACM, Providence,RI, May 1985.

[14] R.E. Tarjan and U. Vishkin. *An Efficient Parallel Biconnectivity Algorithms.* Technical Report, Courant Institute, 1983.

[15] L.G. Valiant. Parallelism in comparison problems. *SIAM J. Comp.*, 4():348–355, 1975.

[16] L.G. Valiant. Universality consideration in vlsi circuits. *IEEE Trans. on Computers*, 30(2):135–140, February 1981.

procedure *Face-vertex-dual*

1. If the Voronoi subgraph H consists of a simple cycle then do
   (a) Pick a point on H and construct a shortest path back to each of the two base cycles of G.
   (b) Using this path and the *BFS* tree in each of the two regions construct a spanning tree of G with radius $O(d\sqrt{k})$
   (c) Find a separator using the algorithm from Theorem 6 in [10].
2. For each region R in G corresponding to a face in the Voronoi Diagram compute the induced *BFS* spanning tree in G from its base cycle.
3. For each R, find all vertices in H on the boundary of R common to 2 more other regions and find the subtrees that span these vertices.
4. Set $G^{\#}$ be the graph consisting of these sub-spanning trees plus the base and boundary cycles.
5. Remove one edge from each base cycle and each boundary cycle in $G^{\#}$.
6. Compute the induced weight of each face.
7. If all face weights are at most 2/3 then do
   (a) Throw out all base cycles that are either not connected or only 1-connected to the rest of the graph.
   (b) Return $G^{\#}$.

   else Let F be the face with weight $> 2/3$.
   (a) Add the edges which were removed in step 5. back to any base cycle internal to F.
   (b) Remove the edges and vertices of G that are either external to F or internal to the base cycles of F.
   (c) Construct a spanning tree from the following subgraphs: F and the induced *BFS* trees of the two regions of the Voronoi diagram that contains the region F.
   (d) Find a separator for this graph using the algorithm from Theorem 6 in [10].

Figure 7: Computing a Dual of the Voronoi Diagram and the Subgraph H of Theorem 3.1.

Determining Edge Connectivity in O(nm)

David W. Matula

Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275

Abstract

We describe an algorithm that determines the edge connectivity of an n-vertex m-edge graph G in O(nm) time. A refinement shows that the question as to whether a graph is k-edge connected can be determined in $O(kn^2)$. For dense graphs characterized by $m = \Omega(n^2)$, the latter result implies that determination of whether a graph is k-edge connected for any fixed k can be accomplished in time linear in input size.

I. Introduction and Summary

Connectivity is an extensively investigated subject in graph theory with applications as varied as reliability, circuit and chip design, and communication networks. The computational questions of determining the edge and vertex connectivities have been investigated by numerous authors [Kl69,ET75,Sc79,Ga80,EH84,LL86]. The best known bound for computing edge-connectivity [EH84,ET75] is $O(n^{5/3}m)$, and derives from the solution of O(n) maximum flow problems in the graph. In this paper we give algorithms for determining the edge-connectivity in O(nm) and for resolving the question of whether or not a graph is k-edge connected in $O(kn^2)$.

In comparison to a source-sink maximum network flow problem, the edge-connectivity problem is easier in that capacities are 0,1, but harder in that a sequence of source-sink flow problems need be solved. Of course other methods of resolving edge-connectivity may be possible, such as the novel approach to vertex-connectivity given in [LL86]. Our improved methods employ traditional flow techniques, with the enhanced efficiency derived from amortizing the cost of each maximum flow problem over a set of vertices between which no further source-sink flow problem need be investigated.

In Section II we give a lemma that will provide the basis of our amortized cost approach, and then a very straightforward algorithm for determining the edge-connectivity in O(nm). A more refined algorithm is given in Section III along with a more detailed analysis showing that the question of k-edge connectedness can be resolved in $O(kn^2)$, with the edge-connectivity then obtainable in $O(\lambda n^2)$. Finally, we give an application of these results to obtain the improved complexity bound that the subgraph H of G having largest edge-connectivity over all subgraphs [Ma72,Ma78] can be found in $O(n^2m)$.

II. Edge Connectivity in O(nm)

Let δ denote the minimum degree and λ the edge-connectivity of the n-vertex m-edge graph $G = (V,E)$. A **dominating set** $S \subset V$ of G has every vertex $v \in V$ either in S or adjacent to a vertex of S. A **cut** $(A,\bar{A})$ denotes the set of all edges between vertex set A and $\bar{A} = V-A$, and a **mincut** then has $|(A,\bar{A})| = \lambda$. Our algorithms have their foundation derived from the following straightforward but fundamental lemma.

Lemma 1: Let G be a graph with a mincut $(A,\bar{A})$ such that $|(A,\bar{A})| = \lambda \leq \delta-1$. Then any dominating set S of G contains vertices of both A and $\bar{A}$.

Proof: For a mincut $(A,\bar{A})$ the sum of the vertex degrees over A satisfies

$$|A|\,(|A| - 1) + \lambda \geq \sum_{v \in A} \deg(v) \geq |A|\delta.$$

so

$$(|A| - \delta)(|A| - 1) \geq \delta - \lambda.$$

By assumption $\lambda \leq \delta - 1$, so then $|A| \geq \delta + 1$, and similarly $|\bar{A}| \geq \delta + 1$. Now at most $\lambda = |(A,\bar{A})| \leq \delta - 1$ vertices of A are incident to edges of $(A,\bar{A})$. Thus some vertex of A is not adjacent to any vertex of $\bar{A}$, so no dominating set can include only members of $\bar{A}$, and similarly no dominating set can include only members of A. □

Our first algorithm utilizes a partition S, T, U of V, where T contains all vertices of V-S adjacent to vertices of S, and U contains all vertices of V-S not adjacent to vertices of S. At

0272-5428/87/0000/0249$01.00 © 1987 IEEE