A Dynamic Algorithm for Maintaining Graph Partitions *

L. G. Aleksandrov
Center of Informatics, Bulgarian Academy of Sciences
G.Bonchev Str. 25-A, 1113 Sofia, Bulgaria

H. N. Djidjev
Department of Computer Science, University of Warwick
Coventry CV4 7AL, UK

Abstract

We propose an algorithm for maintaining a partition of dynamic planar graphs motivated by applications in load balancing for solving partial differential equations on a shared memory multiprocessor. We consider planar graphs of bounded face sizes that can be modified by local insertions or deletions of vertices or edges so that planarity is preserved. In our paper we describe a data structure that can be updated in $O(\log n)$ time after any such modification of the graph, where n is the current size of the graph, and allows an almost optimal partition of a required size to be maintained. More precisely, the size of the separator is within an $O(n^{\delta})$ factor of the optimal for the class of planar graphs, where δ is any positive constant, and can be listed in time proportional to its size. The dynamic data structure occupies $O(n)$ space and can initially be constructed in time linear to the size of the original graph.

1 Introduction

Separator theorems are efficient and widely used tool for the design of efficient divide-and-conquer algorithms. Informally, a separator theorem claims that any graph from a given class can be divided into two or more parts of roughly the same size by removing a small number of vertices. The classical result of Lipton and Tarjan [15] shows that any n-vertex planar graph can be divided into components of no more than $2n/3$ vertices by removing a set of no more than $\sqrt{8n}$ vertices. Moreover, such a separator can be found in $O(n)$ time. Other interesting results include separator theorems for the class of graphs of bounded genus [6, 8, 1], the class of graphs of excluded minor [2], and classes of geometric graphs [17]. Separator theorems have applications in solving efficiently large sparse systems of linear equations [14, 9], for developing algorithms for VLSI layout design [4, 12], for shortest path problems [7], in parallel computing [10], and in computational complexity [16].

A class of problems for whose solutions separator theorems are especially well suited is data partitioning and load balancing for parallel computing. The modern high performance computing systems have large number of processors and their memory is distributed among the processors. In order to achieve high efficiency and speed when using such computers, the

*This work was partially supported by the EPA grant R82-5207-01-0, EPSRC grant GR/M60750, and RTDF grant 98/99-0140. A two-page abstract of this work appeared in the proceedings of CCCG'98.

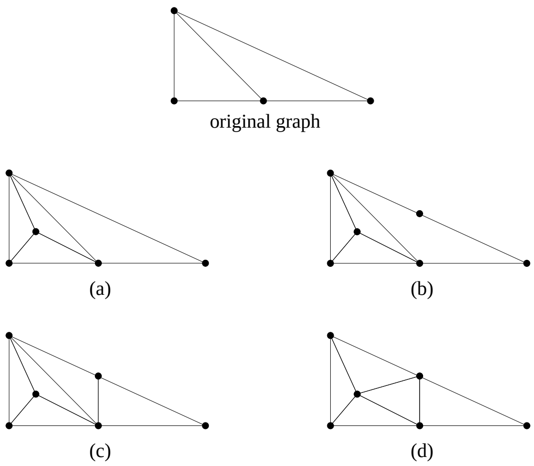

Figure 1: Mesh refinement operations: (a) adding a point inside a face; (b) adding a point onto an edge; (c) adding an edge; (d) edge flipping.

data has to be allocated among the processors so that the computational load is even and the need for communication is minimized. Since this mapping problem is NP-hard, several approaches have been tried to find good approximate solutions. Popular partitioning techniques include Kernighan-Lin's local search algorithm [13], recursive spectral bisection [18], simulated annealing [11], and graph separators [19]. The advantage of the graph-separator approach is that it works well for unstructured meshes with good topology and that it gives a good guaranteed worst-case performance.

In many time-dependent applications, after the initial partition, the data might need to be modified and then reallocated to the processors. Thus the problem for dynamic load balancing arises. For instance, in solving partial differential equations, the mesh might need to be modified (refined or coarsened) after every few time steps. For the case of triangular or bounded face size planar meshes discussed in this paper refinement types of modifications include adding a new point or edge inside an existing face, adding a new point onto an existing edge, edge or face flipping, and others [Figure 1]. Efficient algorithms should use the existing partition in order to compute the new one faster and with a small number of data reallocations.

Unfortunately, there are no known deterministic algorithms that can recompute efficiently the separator after modification of the graph. The existing algorithms need to compute a separator of the new graph from scratch, without using any information from the previous separator. Armon and Reif constructed a dynamic separator algorithm for planar graphs [3], but their algorithm is probabilistic (works in $O(\log^3 n)$ expected time per update) and it needs as an input the so called sphere packing representation [5] of the input planar graph. Although such a sphere representation is known to always exist, it is currently not known whether it is computable in polynomial time.

The main difficulty for designing a fast deterministic algorithm for maintaining partitions of dynamic planar graphs is related to the fact that most known algorithms for constructing separators for static graphs use a breadth-first search as essential step of the computation. There are no known algorithms that can dynamically maintain a breadth-first tree of a planar graph in polylogarithmic time.

In this paper we develop an algorithm that avoids recomputation of breadth-first trees.

Our approach is based on a representation of the current graph as a hierarchy of partitions of different grades for that graph. By handling the modification at an appropriate partition level, we can achieve rebalancing by reallocating a small number of faces. Specifically, we prove that our representation allows a separator decomposition to be recomputed in $O(\log n)$ time after a local insertion or deletion of a vertex or edge that preserves planarity, biconnectivity, and bounded face sizes, where n is the current number of faces of the graph. The total initial partition and the data structure can be computed in $O(n)$ time by using $O(n)$ space.

The paper is organized as follows. In Section 2, we introduce the notation and outline our approach. In Section 3, we specify the requirements to the partitions at each level of the hierarchy and describe efficient algorithm for their construction. The data structure describing the partitions at different levels of the hierarchy and its dynamic maintenance are discussed in Section 4. Finally, in Section 5 we formulate our main theorem and comment the results and their implications.

2 Preliminaries and algorithm outline

Embeddings and regions

A graph is *planar* if it can be embedded in the plane so that no two edges intersect except possibly at a shared endpoint. A graph G already embedded in the plane is called a *plane* graph. *Faces* of G are the maximal connected regions into which the embedding of G divides the plane. There is exactly one infinite face called the *outer* face of the embedding, all other faces are called *internal*. A face can be identified by the cycle of edges on its boundary. By $V(G)$, $E(G)$, and $F(G)$ we will denote the sets of vertices, edges, and faces of G, respectively.

Any set of faces is called a *region* of G. The subgraph that consists of the vertices and the edges incident with the faces of a region R is called a subgraph *induced* by R and will be denoted by $G(R)$. A region R is *connected* if the dual graph of $G(R)$ is biconnected. The maximal connected subregions of R are called *connected components* of R.

An edge e of $G(R)$ is called an *inner edge* of R if both faces incident to e belong to R. Otherwise e is called a *boundary edge*. The boundary ∂R of a region R is the subgraph induced by the boundary edges of R.

A *partition* $\mathcal{R}$ of $F(G)$ is called any set of regions $\{R_1, \cdots, R_r\}$ such that each face of G belongs to exactly one region of $\mathcal{R}$. A partition $\mathcal{R}$ is called *connected* if all of its regions are connected and it is called *weakly connected* if it is either connected or each region has at most two neighboring regions. The *boundary* $\partial\mathcal{R}$ of $\mathcal{R}$ (called also *boundary graph* of $\mathcal{R}$) is the union of the boundaries ∂R_i, $i = 1, \cdots, r$, of its regions.

In the typical case $\partial\mathcal{R}$ may contain long paths of degree 2 vertices. In order to save time and space when dealing with such boundaries, we define a *compressed boundary* $CB(\mathcal{R})$ of $\partial\mathcal{R}$ to be the plane graph resulting after the contraction of any maximal simple path of degree 2 vertices in $\partial\mathcal{R}$ to a single edge and any simple cycle to a triangle. $CB(\mathcal{R})$ is a graph with $|\mathcal{R}|$ internal faces corresponding to the regions of $\mathcal{R}$.

Definition 2.1 *Let G be an n-face plane graph and $\varepsilon > 0$. The partition $\mathcal{R}$ is said to be an* ε-partition *of G if no region of $\mathcal{R}$ has more than εn faces.*

We will make use of the following generalization of a result from [1].

Theorem 1 *Let G be a plane graph with n faces whose maximal size is d and let $h \geq d^2$ be an integer. Then there exists a weakly connected h/n-partition $\mathcal{R} = \{R_1, \cdots, R_r\}$ of G that satisfies the conditions* *(i)* $|\partial R_i| \leq \sqrt{h}$; *(1)*

(ii) $|\mathcal{R}| \leq cn/h$, *(2)*

where $c > 1$ *is a constant.*

In this paper we present an algorithm for maintaining a hierarchy of ε-partitions of a dynamic planar graph G. This algorithm can be used to solve the load balancing problem, since for any integer $p > 1$ one can pick an ε-partition for $\varepsilon \approx 1/p$ and then distribute the regions between the processors using a greedy method so that the sum of the sizes of all regions assigned to a processor does not exceed $n/p + \varepsilon n \leq 2n/p$. If better balancing is required, we just need to reduce the value of ε accordingly.

Our algorithms will handle the following update and query operations. We assume that the original graph is plane with face size bounded by some constant d and that no operation violates planarity or creates a face with size greater than d.

- *insert_vertex(v,e)*: Adds a new vertex v and replace edge $e = (u, w)$ by two edges (u, v) and (v, w).
- *delete_vertex(v)*: Deletes vertex v of degree two and its incident edges (u, v) and (v, w) and adds edge (u, w).
- *insert_edge(u,w,F)*: Adds a new edge (u, w) inside the face F. Assumes u and w are non-adjacent vertices on F.
- *delete_edge(e)*: Deletes edge $e = (u, w)$. Assumes the degrees of u and w are greater than two.
- *list_separator*(ε): Given any $\varepsilon > 0$, lists an εn-separator of the current graph.

Note that these update operations are powerful enough to allow any plane graph to be transformed into any other plane graph using a sequence of update operations.

Our approach

We need to maintain a representation of the graph G that will support fast separator queries, where each query will ask for an ε-partition of the current graph. For that purpose we maintain a balanced tree called a *partition tree* that represents a hierarchy of partitions of G. The partition corresponding to the root defines the coarsest partition of G into at most h regions of roughly the same size, where h will be an appropriately chosen constant . Each subsequent level defines a finer partition, with the leaves corresponding to single faces. Thus, the levels of the partitions tree form a hierarchy of partitions of G. In particular, the leaves of T, on level 0, represent the faces of G, the nodes at the level 1 represent the regions of an h/n-partition $\mathcal{R}^1$, the nodes at level 2 represent regions of an h/n-partition of the graph $CB(\mathcal{R}^1)$, and so on.

When a local change is applied on the graph G, it can affect only nodes that are ancestors of the face where the change occurs. The algorithm maintains a number of local invariants that guarantee the validity of the partitions and the small height of the tree. The algorithm locates the lowest level of T, where some of these invariants is violated (if any). Suppose that at node N any of these invariants is violated. The algorithm considers the subtree consisting of N plus its children and grandchildren and redefines the children of N by finding a new partition of the graph induced by the grandchildren of N. This subtree has $O(1)$ size and thus it can be rebalanced in $O(1)$ time so that all invariants hold. Since the same procedure might need to be applied to all ancestor nodes from N to the root of T, the total update time is proportional to the height of T, which will be shown to be $O(\log n)$.

3 P–tree data structure

In this subsection we define and study the properties of a data structure, called a *P–tree*, for describing partitions of a plane graph G and show that the structure can be constructed in linear time. For the initial construction of the tree, we will define a sequence (hierarchy) of partitions whose elements will be then stored at the nodes of the tree.

More precisely, let G be a plane graph with n faces each of size not exceeding d. Let h be an integer constant such that $h \geq \max(d^2, 2c)$, where c is the constant from Theorem 1. By applying Theorem 1 iteratively, we will construct a sequence of graphs and their h-partitions

$$G = G^{(0)}, \mathcal{R}^{(1)}, G^{(1)}, \cdots, \mathcal{R}^{(l)}, G^{(l)}$$

called a *GR-sequence*, so that the graph $G^{(0)}$ is the original graph G and the region partition $\mathcal{R}^{(l)}$ consists of a single region of no more than h faces.

Constructing the GR-sequence

Assuming that $G^{(i-1)}$ has already been constructed for some $i > 1$ and that it has $n_{i-1} > h$ faces with each face boundary of at most $\sqrt{h}$ edges, we can construct $G^{(i)}$ from $G^{(i-1)}$ by applying the following procedure. We let $G^{(0)} = G$.

Algorithm GR–SEQUENCE

Step 1: Apply Theorem 1 with a parameter h on $G^{(i-1)}$ to find an h/n_{i-1}-partition $\mathcal{R}^{(i)}$ of $G^{(i-1)}$ such that $|\mathcal{R}^{(i)}| \leq cn_{i-1}/h$.

Step 2: For each region $R \in \mathcal{R}^{(i)}$ construct the subgraph $G^{i-1}(R)$ of G^{i-1} induced by R and find its boundary ∂R.

Step 3: If $\mathcal{R}^{(i)}$ is a connected partition, then $G^{(i)}$ will just be the compressed boundary of $\mathcal{R}^{(i)}$. In the general case, replace each region of $\mathcal{R}^{(i)}$ by a single face of $G^{(i)}$ as follows.

3.1. Compute the compressed boundary graph $CB = CB(\mathcal{R}^{(i)})$ of $\mathcal{R}^{(i)}$. For each new edge e defined during the compression store the list of edges of the path corresponding to e.

3.2. For any non-connected region R_{nc} of $\mathcal{R}^{(i)}$ merge the faces in CB corresponding to the connected components of R_{nc} into one of those faces (arbitrarily selected).

Note that the above algorithm constructs $G^{(i)}$ together with its embedding in the plane and its faces correspond to the regions of $\mathcal{R}^{(i)}$. According to Theorem 1 $G^{(i)}$ has no more than cn_{i-1}/h faces and the size of each face does not exceed $\sqrt{h}$. Thus the same algorithm can be used for constructing $G^{(i+1)}$ assuming $G^{(i)}$ has at least h faces.

Since the parameter h was chosen to be at least $2c$, the graph $G^{(l)}$ will be obtained in no more than $\log n + 1$ iterations. Iteration k takes time proportional to the size of $G^{(k-1)}$ and thus the total time for constructing the GR-sequence will be

$$\sum_{k=0}^{l-1} O((c/h)^k n) = O(n).$$

Constructing the P–tree

Let G be a plane graph for which a GR-sequence S has already been constructed. A *P–tree* for G with respect to S is a data structure whose elements are associated with a rooted tree $T_h(G)$ with $l+1$ levels. The k-th level of $T_h(G)$ for $k = 1, \cdots, l$ contains the information gathered during the k-th iteration of the construction of S, namely, the partition $\mathcal{R}^{(k)}$, its boundary $\partial\mathcal{R}^{(k)}$, and the graph $G^{(k)}$. The nodes at the k-th level of the P–tree correspond to the regions of $\mathcal{R}^{(k)}$ (or, equivalently, to the faces of $G^{(k)}$). The leaves of $T_h(G)$ are at level 0 and each leaf corresponds to a face of the original graph $G = G^{(0)}$. The root node is at level l and contains the graph $G^{(l)}$.

With each node N on level $k > 0$ we associate the following information:

(i) $F(N)$ – the face of $G^{(k)}$ corresponding to N;

(ii) $\mathcal{R}(N)$ – the region of $G^{(k-1)}$ corresponding to $F(N)$; and

(iii) $\partial\mathcal{R}(N)$ – the boundary graph of $\mathcal{R}(N)$.

If N is a node on level 0, then $F(N)$ is defined as in (i) above and $\mathcal{R}(N) = \partial\mathcal{R}(N) = F(N)$. Let $G(N)$ denote the graph $G(\mathcal{R}(N))$.

Note that the amount of data associated with any node N of T is proportional to the size of $G(N)$. Since $G(N)$ is a planar graph with no more than h faces and each face has size at most $\sqrt{h}$, the size of $G(N)$ is $O(h^{3/2})$ (which is a constant since $h = O(1)$).

The edges of $T_h(G)$ (called *links* hereafter to be distinguishable from the edges of $G^{(k)}$) connect certain pairs of nodes on consecutive levels. More precisely, there is a link between a node N from level $k-1$ corresponding to a region R, and a node N' from level k corresponding to a region R', iff R is transformed by Algorithm GR-SEQUENCE into a face of R'. By Theorem 1, each node of $T_h(G)$ has at most h children.

Note that any edge of $G^{(k-1)}$ appears in either three or four nodes of the tree $T_h(G)$. It appears twice on the $(k-1)$-th level at the nodes that correspond to its two incident faces and also at one or two nodes on the k-th level, depending on whether this edge is entirely inside some of the regions of $\mathcal{R}^k$, or lies between two such regions. We assume that there are pointers between any edge and its occurrences on the lower level. These pointers define an *edge forest* EF on the set of edges of $G^{(k)}$ for $k = 0, \cdots, l$ that is consistent with $T_h(G)$. Namely, all edges on path that is compressed to an edge e are descendants of e. Thus, the edges of the graph $G^{(k)}$ for $k > 1$ have descendants that are edges of $G^{(k-1)}$. On the other hand, an edge e' from level $k-1$ has an ancestor if and only if e' is on the boundary of some of the regions of ∂R^k.

In accordance with EF, we define *weights* $wt(\cdot)$ on the edges of $G^{(k)}$ as the number of their descendants in F at level 0. The weight of the boundary of a region or a face is defined as the sum of the weights of the edges on that boundary.

Definition 3.1 *A P–tree node N is called* balanced, *if all of the following three conditions are satisfied.*

(B1) N and each child of N have no more than h children each.

(B2) The ratio between the number of the children and the number of the grandchildren of N does not exceed c/h, where c is the constant from Theorem 1.

(B3) For any child N_1 of N $wt(\partial F(N_1)) \leq dh^{(k-1)/2}$, where k is the level of N.

A P-tree is called balanced *if all its non-root nodes are balanced.*

By Algorithm GR–SEQUENCE and Theorem 1 we have the following.

Lemma 3.1 *If G is a plane graph with n faces, then the algorithm from this section constructs in $O(n)$ time a balanced P–tree for G.*

Proof: Since Step 1 of Algorithm GR–SEQUENCE constructs an h/n_{i-1}-partition of $G^{(i-1)}$, any node of the tree has at most h children, implying (B1). Property (B2) follows from conditions (1) and (2) of Theorem 1. Finally, Property (B3) follows from condition (1) of Theorem 1, the assumption that the maximum face size of the original graph is no more than d, and an induction on k. ■

We will maintain our P–tree balanced.

3.1 Update and query operations

Next we describe the basic operation on P–trees used to maintain the balance property of nodes. It is possible that more than one node is unbalanced at a time, but all unbalanced nodes belong to a single simple path from certain node to the root of the tree.

Balancing the tree

We will first describe the basic algorithm for maintaining the balance of the P–tree, called *Fix-Tree*.

Algorithm FIX-TREE starts at an unbalanced node N on a lowest possible level and makes the subtree consisting of N and all its descendants balanced in constant time. Then the same operation is applied on the parent of N, and so on, until the root is processed.

Let N be a node at level k, $2 \leq k \leq l+1$, that is unbalanced, but all its proper descendants are balanced. Denote the subgraph of $G^{(k-2)}$ induced by the faces of the grandchildren of N by $G^{(k-2)}(N)$. Repartition $G^{(k-2)}(N)$ using the algorithm from Theorem 1 and construct again the portion of the tree (with height two) rooted at N, its children, and its grandchildren using algorithm similar to the one for construction of P–trees. The time required to fix N will be proportional to the number of faces of $G^{(k-2)}(N)$, which is $O(h^2)$ by of Property (B1) of Definition 3.1. Since the height of the P–tree is $O(\log n)$, balancing the tree will take $O(\log n)$ time.

Implementation of the update operations

We will describe and analyze the implementation of the four update operations described in Section 2. Our assumption is that the current graph G is a connected plane graph with no vertices of degree 1 and no face size exceeding d and that the update operations preserve these properties. By T we denote the current P-tree of G.

insert_vertex(v, e) The operation asks that edge $e = (u, w)$ be replaced by edges $e_1 = (u, v)$ and $e_2 = (v, w)$.

Let F_1 and F_2 be the two faces of G incident to e and let N_1 and N_2 be the corresponding leaf nodes in T. First, update at nodes N_1 and N_2 the corresponding descriptions of F_1 and F_2 by deleting e and inserting e_1 and e_2 into the corresponding doubly linked lists. Update the weights of all edges that were ancestors of e in the edge forest EF associated with T. Since these updates can change the weights of the faces associated with the ancestors of N_1 and N_2, we need to apply *Fix_Tree* algorithm on the lowest unbalanced ancestors of each of N_1 and N_2.

delete_vertex(v) This operation deletes a vertex v of degree two, thereby changing the common boundary of the two faces incident to v. Hence, *delete_vertex* can be implemented in a similar way as *insert_vertex* by changing the information associated with the two leaf nodes representing the faces incident to v and updating the weights of $O(1)$ edges per each level of the edge-forest associated with T.

insert_edge(u, w, F) Given two non-adjacent edges u and w on the same face F of G, *insert_edge* adds a new edge $e = (u, w)$ inside F.

Denote by F_1 and F_2 the faces into which e splits F. Delete the leaf node N that corresponds to F and create two new nodes N_1 and N_2 representing the new faces. Make N_1 and N_2 children of the parent PN of N. Next, update the region subgraph stored at PN by adding to $\mathcal{R}(PN)$ the edge e inside face F. Finally, execute algorithm *Fix_Tree* on the lowest unbalanced ancestor of PN.

delete_edge(e) This operation deletes edge $e = (u, w)$, assuming the degrees of u and w are greater than two.

Let F_1 and F_2 be the two faces of G incident to e and let N_1 and N_2 be the corresponding nodes in T. Let F be the face that results after the deletion of e and the merge of F_1 and F_2. In order to implement that operation, we delete N_1 and N_2 and create a new leaf node N representing the face F.

Assume that N_1 and N_2 have different parents, say PN_1 and PN_2. In this case we delete N_2 and replace N_1 with the new node N. As a result, the number of children of PN_2 is reduced by one and one of the children of PN_1 represents a larger region. We make the corresponding changes in the parents PN_1 and PN_2. These include updates in the structures $F(PN_i)$, $\mathcal{R}(PN_i)$, and $\partial\mathcal{R}(PN_i)$, $i = 1, 2$. In particular, e is deleted from both faces $F(PN_1)$ and $F(PN_2)$ and all edges of F_2. We continue to make similar changes on the ancestors of $F(PN_1)$ and $F(PN_2)$ until the nearest common ancestor of N_1 and N_2 is reached.

Finally, we run the Algorithm *Fix_Tree* on the parents of N_1 and N_2.

If in result of some update operation the degree of the root R becomes one, then we cut R. In case the degree of R becomes greater than $h+1$, then we define a new vertex R' to be a parent of R and run Algorithm *Fix_Tree* on vertex R'.

Note that any of the update operations described above changes information at only constant number of nodes at any level of T and that we applied *Fix_Tree* on only one or two nodes of T. If all nodes of T are balanced, then by Condition (B2) the height of T is at most $\log_{h/c} n \leq \log n$, since we have chosen $h \geq 2c$. Hence the time complexity of Algorithm *Fix_Tree* is also $O(\log n)$. Thus we have the following lemma.

Lemma 3.2 *A balanced P-tree representing a planar connected graph G with no vertex of degree one can be maintained subject to the operations* insert_vertex, delete_vertex, insert_edge, *and* delete_edge *in $O(\log n)$ time per operation, assuming the maximum face size never exceeds a constant d.*

Extraction of an ε-partition

Recall that any node N_i at the k-th level represents a face $F(N_i)$ of $G^{(k)}$. Similarly, the children of N_i correspond to the faces of $\mathcal{R}(N_i)$ (the region of $G^{(k-1)}$ associated with $F(N_i)$), and so on. Therefore, N_i defines a region in the graph $G = G^{(0)}$ represented by the leaves that are descendants of N_i. We denote that region of G by $\bar{R}(N_i)$. Let $\mathcal{R}(G,k)$ denotes the partition $\{\bar{R}(N_1),\cdots,\bar{R}(N_s)\}$ of G, where $\{N_1,\cdots,N_s\}$ is the set of all nodes on level k.

Lemma 3.3 *Let $1 \le k \le l$ and $\varepsilon_k = h^k/n$, where n is the number of faces in G. Then the partition $\mathcal{R}(G,k)$ is an ε_k-partition of G with boundary of size not exceeding $d\sqrt{\frac{n^{1+\log c/\log(h/c)}}{\varepsilon_k}}$.*

Proof: Clearly, $\mathcal{R}(G,k)$ is a partition of G since any face of G (which is a leaf of the P-tree of G) belongs to exactly one of the regions $\bar{R}(N_i)$. For the size of the regions of that partition $\mathcal{R}(G,k)$ we have

$$|\bar{R}(N_i)| \le h^k = \varepsilon_k n, \text{ for } i = 1,\cdots,s,$$

since the maximum degree of T is no more than $h+1$. For the size of the boundaries $\partial\bar{R}(N_i)$ we have

$$|\partial\bar{R}(N_i)| = wt(F(N_i)) \le dh^{k/2},$$

according to Condition (B2) of Definition 3.1. The number of nodes at level k is at most h^{l-k}, where $l+1$ is the number of the levels in T. Therefore, the total weight of the boundary of $\mathcal{R}(G,k)$ is

$$\begin{aligned} wt(\mathcal{R}(G,k)) &\le dh^{k/2}h^{l-k} = dh^{l-k/2} \le d\frac{h^{\log n/\log(h/c)}}{h^{\frac{k}{2}}} \\ &= d\,2^{\frac{\log n \log h}{\log(h/c)}}/\sqrt{\varepsilon_k} = d\sqrt{n^{1+\log c/\log(h/c)}/\varepsilon_k}. \end{aligned}$$

■

Now, in order to process a query asking for an ε-partition of G for some $\varepsilon \in (0,1)$, we determine the level k for which $h^{k-1} < \varepsilon n \le h^k$ and apply Lemma 3.3. Let us estimate the time necessary to list the boundary of the resulting partition. Recall that $F(N)$ is a compressed image of the boundary $\partial\mathcal{R}(N)$ of the region $\mathcal{R}(N)$ of $G^{(k-1)}$. Each edge of $F(N)$ represents a path on $\partial\mathcal{R}(N)$. Each edge of $\partial\mathcal{R}(N)$ itself represents a path in $G^{(k-2)}$ (if $k \ge 2$), and so on. Therefore, each edge of $F(N)$ can be traced down to a path in the original graph $G = G^{(0)}$. The boundary $\partial\bar{R}(N)$ of the region $\bar{R}(N)$ can be extracted from the P-tree in time $O(\partial\bar{R}(N))$. Thus the boundary of the partition can be listed in time proportional to its size, which is $O(\sqrt{n/\varepsilon})$. Thus we have the following lemma.

Lemma 3.4 *The k-th level of a P-tree represents a h^k-partition of G whose boundary B can be listed in $O(|B|)$ time.*

We summarize the above results in the following theorem.

Theorem 2 *Let G be a plane graph with n faces. Then for any $\delta > 0$ a data structure exists that*

(i) can be constructed in $O(n)$ time;

(ii) supports operations insert_vertex, delete_vertex, insert_edge *and* delete_edge *in $O(\log n)$ time assuming the maximum face size of any intermediate graph is $O(1)$.*

(iii) supports operation list_separator (ε) *for any $\varepsilon > 0$ in time proportional to the separator's size, which is bounded by $O(\sqrt{n^{1+\delta}/\varepsilon})$.*

Proof: Follows from Lemmas 3.1–3.4 for $h = c^{1/\delta}$. ■

References

[1] Lyudmil Aleksandrov and Hristo N. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal on Discrete Mathematics*, 9:129–150, 1996.

[2] Noga Alon, Paul Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. *Proceedings of the 22nd Symp. on Theory of Computing*, pages 293–299, 1990.

[3] D. Armon and J. Reif. A dynamic separator algorithm. In *Proc. 3rd Worksh. Algorithms and Data Structures*, pages 107–118. Lecture Notes in Computer Science 709, Springer-Verlag, Berlin, 1993.

[4] S.N. Bhatt and F.T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28:300–343, 1984.

[5] J.H. Conway and N.J.A. Sloane. *Sphere Packings, Lattices and Groups.* Springer-Verlag, 1988.

[6] Hristo N. Djidjev. A separator theorem. *Compt. rend. Acad. bulg. Sci.*, 34:643–645, 1981.

[7] G.N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.

[8] John R. Gilbert, Joan P. Hutchinson, and Robert E. Tarjan. A separator theorem for graphs of bounded genus. *J. Algorithms*, 5:391–407, 1984.

[9] John R. Gilbert and Robert E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1987.

[10] Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Proceedings of 24th Symp. on Theory of Computing*, pages 507–516, 1992.

[11] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, pages 671–680, 1983.

[12] C.E. Leiserson. Area efficient VLSI computation. In *Foundations of Computing*. MIT Press, Cambridge, MA, 1983.

[13] B.W. Kernighan & S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal*, 49:291–308, 1970.

[14] Richard J. Lipton, D. J. Rose, and Robert E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.

[15] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math*, 36:177–189, 1979.

[16] Richard J. Lipton and Robert E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.

[17] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. *Proceedings of the 32nd FOCS*, pages 538–547, 1991.

[18] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, July 1990.

[19] Eric J. Schwabe, Guy E. Blelloch, Anja Feldmann, Omar Ghattas, John R. Gilbert, Gary L. Miller, David R. O'Hallaron, Jonathan R. Shewchuk, and Shang-Hua Teng. A Separator-Based Framework for Automated Partitioning and Mapping of Parallel Algorithms for Numerical Solution of PDEs. In *Proceedings of the First Annual Summer Institute on Issues and Obstacles in the Practical Implementation of Parallel Algorithms and the Use of Parallel Machines in Parallel Computation (DAGS/PC '92)*, pages 48–62. Dartmouth Institute for Advanced Graduate Studies, June 1992.