DOI: 10.1007/s12541-011-0056-1

A Divide-and-conquer Delaunay Triangulation Algorithm with a Vertex Array and Flip Operations in Two-dimensional Space

Sang-Wook Yang[1], Young Choi[1,#] and Chang-Kyo Jung[1]

1 School of Mechanical Engineering, Chung-Ang University, 221 Heukseok-dong, Dongjak-gu, Seoul, Korea, 156-756
Corresponding Author / E-mail: yychoi@cau.ac.kr, TEL: +82-2-820-5312, FAX: +82-2-817-5101

KEYWORDS: Delaunay triangulation, Divide-and-conquer, Flip-based merge, Geometric algorithms, Graphics data structure

Divide-and-conquer algorithms provide computational efficiency for constructing Delaunay triangulations; however, their implementation is complicated. Most divide-and-conquer algorithms for Delaunay triangulation utilize edge-based structures, because triangles are frequently deleted and created during the merge process. However, as our proposed divide-and-conquer algorithm does not require existing edges to be deleted in the merge process, a simple array-based data structure can be used for the representation of the triangulation topology. Rather than deleting the edges of the conflicting triangles, which was used in previous methods, the Delaunay property is also preserved with a new flip propagation method in the merge phase. This array-based data structure is much simpler than the commonly used edge-based data structures and requires less memory allocation. The proposed algorithm arranges sites into a permutation vector that represents a kd-tree with an array; thus, the space partitioning information is internally represented in the array without any additional data. Since the proposed divide-and-conquer algorithm is compact, the implementation complexity of conventional divide-and-conquer triangulation algorithms can be avoided. Despite of the simplicity of this new algorithm, the experimental results indicate that the computational efficiency is comparable to the previous divide-and-conquer algorithms.

Manuscript received: November 30, 2010 / Accepted: January 31, 2011

1. Introduction

The divide-and-conquer (D&C) strategy is one of the most popular methods for constructing Delaunay triangulation from a set of points. In the D&C algorithm, a set of sites is divided into small subsets, and the Delaunay triangulations of the subsets are subsequently merged into a global triangulation.

Shamos and Hoey originally proposed a D&C algorithm for the construction of Voronoi diagrams, showing that the time complexity is $O(n\log n)$ in the two-dimensional space.[1] Lee and Schachter proposed a direct calculation for Delaunay triangulation using the D&C algorithm.[2] Guibas and Stolfi devised a quad-edge-based data structure and related operations so as to manipulate the topologies of Voronoi diagrams and described the algorithm in terms of their proposed operations.[3] In earlier research, D&C algorithms divided the space along a single axis of the coordinate system. Later, multi-dimensional space partitioning was applied in order to improve the robustness and efficiency of the calculations in the triangulation algorithms. Ohya et al. and Maus proposed expected linear time algorithms by applying a bucketing technique in two-dimensional space.[4,5] Dwyer modified Guibas and Stolfi's algorithm by dividing $n/\log n$ cells from n sites in the two-dimensional space and reduced the expected time to $O(n \log\log n)$.[6] Katajainen and Koppinen divided two-dimensional space with a quad-tree data structure and proposed an $O(n)$ expected time algorithm.[7] Lemaire et al. adaptively divided the two-dimensional space with a balanced kd-tree and estimated a general probabilistic result for higher dimensional spaces.[8]

The new D&C algorithm proposed in this paper divides two-dimensional spaces into kd-tree partitioning regions, as in Lemaire's algorithm; however, the proposed algorithm does not construct an actual kd-tree, but rather arranges the set of sites into a permutation vector of a vertex array.[9] This array represents the kd-tree without needing any additional data structure, since the partitioning information is internally represented as a sequence of vertex elements. The new merge process is based on edge flip operations without deleting any of the existing triangles or edges. This is the most significant feature of the new algorithm, differentiating it from

the previously devised D&C algorithms. Since there are no edge deletions and no necessary bookkeeping of the complex intermediate states for topology changes, our algorithm can be easily implemented in a compact manner. The new D&C algorithm was named as Flip-Based D&C (FBD&C).

The algorithm overview and the contributions of this paper are briefly discussed in Section 2. A detailed explanation of the data structure and space partitioning is discussed in Section 3. Finally, an empirical analysis of the algorithm is discussed in Section 4.

2. Algorithm Overview

2.1 Conventional divide-and-conquer Delaunay triangulations

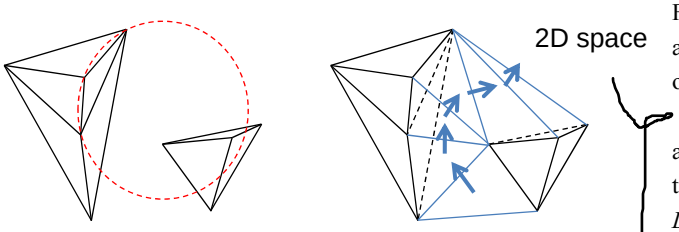
A divide-and-conquer algorithm usually starts with the points that are sorted along at least one axis of the space. If $|V|$, the number of points in V, is three or less, the Delaunay triangulation $T(V)$ for the point set V is directly calculated; otherwise, the point set V is divided into approximately identically sized subsets - V_L on the left side and V_R on the right side. Then, the sub triangulations $T(V_L)$ and $T(V_R)$ are recursively calculated and merged into the resulting triangulation $T(V)$, so that $T(V)$, $T(V_L)$, and $T(V_R)$ are all Delaunay triangulations.

The conventional D&C algorithms determine whether the triangles $T(V_L)$ and $T(V_R)$ satisfy the Delaunay property after the triangulations have been merged (Fig. 1 (a)). The triangles that violate the Delaunay property are deleted, and the remaining triangles are then merged (Fig. 1(b)). In Figure 1 (b), the removed edges from $T(V_L)$ and $T(V_R)$ are represented as dotted lines. Most of the previous algorithms utilize an edge-based data structure, such as a quad-edge data structure, in order to handle topological relationships during the frequent edge deletion and creation that occurs in the merge process.[2,3,10-12]

2.2 Contribution of the proposed algorithm

The proposed algorithm has two main characteristics that distinguish it from previous D&C triangulation algorithms.

First, there is no explicit in-circle test in the merge process. The conventional merge process, discussed in Section 2.1, consists of two phases. The first is the search and deletion of edges that would become invalid in the merged triangulation, and the other is the sewing of two sub-triangulations (See Fig. 1). In the conventional D&C Delaunay triangulation algorithms, if the circumcircle of a triangle $t_L \in T(V_L)$ contains a vertex $v_R \in V_R$ when the two sub Delaunay triangulations $T(V_L)$ and $T(V_R)$ are merged into the resulting triangle $T(V)$, then the triangle t_L must be removed before the merge. Some researchers attempted to improve the efficiency of the in-circle test scheme.[10,11] Conversely, the proposed FBD&C algorithm bypasses the process of searching for conflict triangles and removing them. Instead of a searching and removing scheme, a flip operation, which is generally used in incremental approaches, is applied during the merge process to keep the Delaunay property of the intermediate and resulting triangulations. Since the flip operation does not delete any existing edges, but rather modifies local topological relationships, complex and detailed records of the intermediate topological states are not necessary. In the proposed approach, instead of using time to search for conflict edges, flipping time is required; thus, the number of flip occurrences becomes an important factor in determining algorithm efficiency. Although it is not easy to completely prove algorithm efficiency in an asymptotic manner, our experiments showed that the flip occurrences linearly increase with the number of sites to be merged. The algorithm details are discussed in Section 3.3, and the experimental results are described in Section 4.



(a) Test empty circle property (b) Delete and merge

Fig. 1 Conventional merge process

The second main characteristic is that the proposed algorithm uses a simple array-based triangle data structure. Although an array-based data structure is widely used in incremental construction algorithms for its compactness, previous D&C triangulation algorithms utilized edge-based data structures so that the various edge-based topological operations could be performed in the merge process. However, since the existing edges in the FBD&C are not deleted, simple arrays of vertices and triangles are sufficient to process the topological operations for the proposed triangulation. The data structure consists of only two arrays - a vertex array and a triangle array. A triangle is represented as three indices for vertices, and a vertex consists of the coordinates and only one reference to one of the connected triangles. The vertex array is sorted along an optimized kd-tree order so that implicit space partitioning information is available without any additional data. The details are discussed in the following sections.

3. Flip-based divide-and-conquer triangulation

3.1 Indexed triangle data structure

In a previous study,[14] Yang et al. proposed the Constraint Embedded Triangles (CET) data structure for the sequential constrained triangulation of trimmed surfaces. The data structure used by the FBD&C is almost the same as the CET data structure, except for the constraint information. This is due to the fact that the FBD&C algorithm does not handle constraint edges. In this section, a non-constraint version of the CET data structure and the operations for manipulating the triangle data are briefly introduced.

Triangulation in two-dimensional space can be represented with an array of n vertices $V = \{v_0, v_1, ..., v_{n-1}\}$ and an array of m triangles $T = \{t_0, t_1, \ldots, t_{m-1}\}$. A vertex in V is represented as $v = (p, L)$, where, p represents the coordinates of the vertex and L is an array index of a triangle in T connected to v. A triangle t in T is denoted as $t = (M_0, M_1, M_2, N_0, N_1, N_2)$, where M_i represents an

array index of a vertex in $\boldsymbol{V}$, and N_i represents the array index of a triangle in $\boldsymbol{T}$. The vertices referenced by M_0, M_1, and M_2 occur in counter-clockwise order. The triangle referenced by N_i is a neighbor to triangle t, which is not connected to the vertex referenced by M_i. The neighbors of triangle t are denoted as *tri(t, k)*, where k is the index corresponding to a particular neighbor (k=0,1,2). Likewise, *vert(t, k)* indicates the vertex of the specified index. Since each vertex points to exactly one triangle, *tri*(v) is used to reference that unique triangle. When vertex v_a is referenced by a triangle t or triangle t_a, which is the triangle adjacent to t, the operations that determine the indices in triangle t are defined as *index*(t, v_a) and *index*(t, t_a).

Functions?

Although the data structure does not explicitly define the edges, an edge shared by a triangle t and *tri*(t, i) can be denoted as *edge*(t, i). Some abstract operations, such as *edge*, are adopted in this paper for clear and compact description of the algorithm. For example, the authors defined a flip operation for an *edge*(t, i), as *flip*(t, i). The flip operation changes the edge shared by two adjacent triangles into the one, which joins their un-shared vertices. However, since there is no explicit edge data in the proposed data structure, the flip operation is performed by modifying the topological relationships of the two triangles sharing *edge*(t, i) and their adjacent triangles. Figure 2 shows the topology change of the two adjacent triangles before and after the *flip*(t, i) operation. The adjacent triangle t_1 and index i_1 can be found using t_1=*tri*(t, i) and i_1=*index*(t_1, t) in the left figure, and the flip result shown on the right is achieved only by changing the reference indices of the adjacent entities. Since there is no deletion or creation of triangles during the *flip* operation, the topological consistency can be easily maintained.

3.2 Space partitioning with kd-tree in array form

The proposed FBD&C algorithm recursively divides two-dimensional space into small regions with the *k*d-tree. Each node of the *k*d-tree has references to its left and right children and sometimes needs a discriminator index for fast traversal.[15] However, if an array is used instead of a tree data structure, the indices for each node are not necessary, and, thus, memory space for two or three indices for each data node is saved. Bentley introduced a permutation vector,[9] which utilized a *k*d-tree in an array form for this purpose. The FBD&C algorithm divides a two-dimensional space by building a balanced permutation vector from a site array.

Space partitioning with the permutation vector is defined as a problem that calculates a sorted array $\boldsymbol{V}$ [v_0, v_1, …, v_{n-1}] from a k-tuple data vector $\boldsymbol{S}$ [s_0, s_1, …, s_{n-1}], where $s_i \in \boldsymbol{R}^k$. Since the elements are only rearranged in sequence, the data set does not change, i.e. {v} = {s}. The following algorithm sort*k*d is a variation of Bentley's optimization algorithm of the *k*d-tree,[16] modified so that it could be applied to an array data structure. The algorithm recursively performs partial sorting and then calculates a balanced permutation vector from an arbitrarily arranged vertex array. The index j is the discriminator for the depth, and the j-median is a median element determined by the j-th data elements of the k-tupple data. The terms j-less and j-greater represent the results of a comparison between the j-th data elements.

A binary search tree

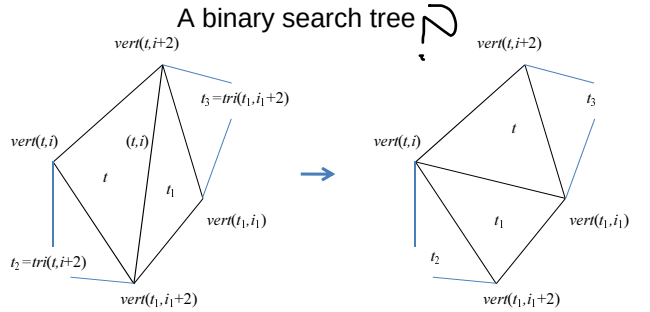


Fig. 2 Topology change in a *flip* operation

Algorithm sort*k*d

input: Array $\boldsymbol{V}[v_p, v_{p+1}, \ldots, v_{r-1}, v_r]$, depth j, and indices p and r

output: *k*d-sorted array $\boldsymbol{V}$

Selecting Pivot

Step 1: Set index $q = (p+r) / 2$. if $p \geq r$ then quit.

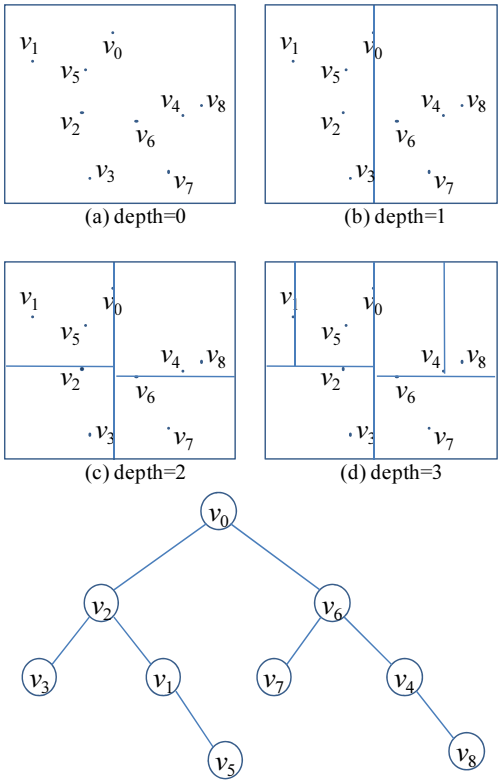
Step 2: Select j-median element v_m then swap v_q with v_m.

Step 3: Move all j-less elements than v_m to $\boldsymbol{V}[v_p, \ldots, v_{q-1}]$ and j-greater elements than v_m to $\boldsymbol{V}[v_{q+1}, \ldots, v_r]$.

Step 4: Do sort*k*d($\boldsymbol{V}[v_p, \ldots, v_{q-1}], j+1$) and sort*k*d($\boldsymbol{V}[v_{q+1}, \ldots, v_r], j+1$).

Since each step in *Step* 1~*Step* 2 is bound to $O(n)$, and the number of elements is reduced by half for each depth, the total time required for the algorithm is $O(n\log n)$.[16]

The figures in Fig. 3 show the space partitioning process for a vertex array $\boldsymbol{V}$= [v_0,v_1,…,v_8] in two-dimensional space. The partitioning is recursively repeated until there are fewer than three vertices in a region. Figures 3(b) and (c) show the intermediate



(e) *k*d-tree representation of the array $\boldsymbol{V}_a$

Fig. 3 Space partitioning with a permutation vector

states in the partitioning process. This partitioning process is similar to the construction scheme of a balanced *k*d-tree,[16] but there is no actual construction of the *k*d-tree. Instead, only the sequence of the vertices is rearranged in order to represent the *k*d-tree, and the rearranged vertex array is called a permutation vector. The final state shown in Figure 3(d) is represented as the rearranged array $\boldsymbol{V}_a$ = [v_3, v_2, v_1, v_5, v_0, v_7, v_6, v_4, v_8], and the array sequence is the same as the in-order traversal sequence in the balanced *k*d-tree from the given vertices set (See Fig. 3(e)).

The divided vertex sets in Fig. 3(b) can be denoted as $\boldsymbol{V}_{a1L}$ and $\boldsymbol{V}_{a1R}$. They correspond to the two parts divided by *x*-median in $\boldsymbol{V}_a$ (i.e. $\boldsymbol{V}_{a1L}$ = [v_3, v_2, v_1, v_5, v_0] and $\boldsymbol{V}_{a1R}$ = [v_7, v_6, v_4, v_8]). If we divide $\boldsymbol{V}_{a1L}$ by the *y*-median, the subsets become $\boldsymbol{V}_{a1L2B}$ = [v_3, v_2, v_1] and $\boldsymbol{V}_{a1L2T}$ = [v_5, v_0], and they correspond to the left bottom and top regions (shown in Fig. 3(c)).

3.3 Flip-based merge algorithm

3.3.1 Merging triangulations

Divide-and-conquer algorithms generate triangles from small regions and then merge the local triangulations into a global triangulation. Figure 4 shows the merge process performed in both the *x* and *y* directions. Figure 4(a) shows the initial elementary triangulations with the regions partitioned along an optimized *k*d-tree, as in Lemarie's method.[8] Figures 4(b) and (c) show the intermediate states of the merge process, and Fig. 4(d) shows the resulting triangulation.

3.3.2 The proposed merge algorithm

As discussed in the section 2.2, the proposed merge algorithm performs sewing sub triangulations and empty circle test in single stage. When triangle *t* is generated between $\boldsymbol{T}(\boldsymbol{V}_L)$ and $\boldsymbol{T}(\boldsymbol{V}_R)$, an empty circle test for the edge shared by *t* and other existing triangles is performed, and edge-flips occur if necessary. Edge flipping in the FBD&C is similar to that of incremental construction[17] but is slightly more complicated.

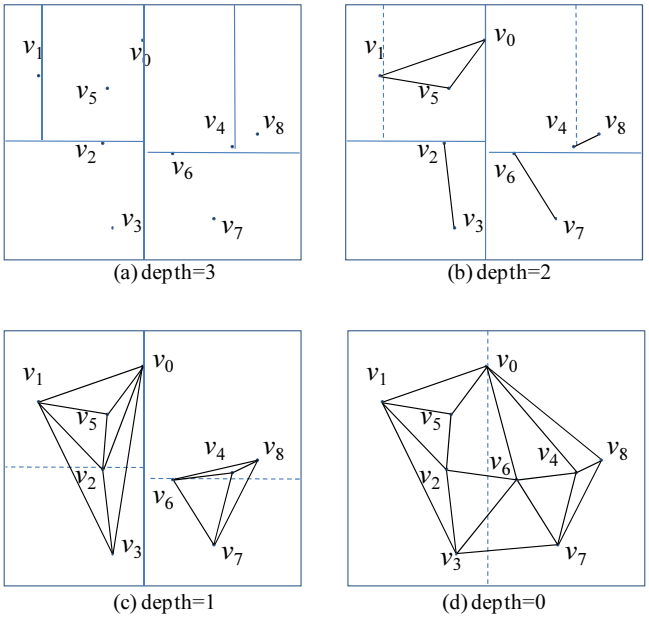


Fig. 4 Merging triangulations

Figure 5 shows an example of the merge process in the FBD&C. As seen in Fig. 5(a), two new triangles are created between $\boldsymbol{T}_L(\boldsymbol{V}_L)$ and $\boldsymbol{T}_R(\boldsymbol{V}_R)$. The second new triangle violates the empty circle property with respect to vertices in the triangulation $\boldsymbol{T}_L(\boldsymbol{V}_L)$; thus, an edge flip occurs (Fig. 5(b)). In Fig. 5(c), one of the flipped triangles violates the empty circle property again; therefore, a second flip is necessary. Flip operations are propagated until all of the triangles around the newly created triangle satisfy the empty circle property, such that the merged triangulaion remains a Delaunay triangulation. Although the number of propagations for an edge in an incremental construction does not exceed six in the case of Delaunay triangulation in two-dimensional space,[4] the counts of flip propagation in the merge phase of the FBD&C are not necessarily predictable. However, the experimental results, which will be discussed in a later section, show that the number of flip occurrences linearly increased along the number of sites in the triangulation.

The algorithm *delaunay2d* describes the overall process of the FBD&C, which recursively merges sub-triangulations in two-dimensional directions. Although the algorithm is similar to Lemaire's algorithm,[8] which performs a two-dimensional merge, the proposed algorithm uses the same *merge* algorithm regardless of partitioning directions.

Algorithm *delaunay2d*
input: Set $\boldsymbol{V}$={v_p,…, v_r}, discriminator *j*
output: $\boldsymbol{DT}(\boldsymbol{V})$
Step 1: If $|\boldsymbol{V}| \le 3$ and the vertices are not collinear,
then $\boldsymbol{DT}(\boldsymbol{V})$ = *elementary-delaunay*($\boldsymbol{V}$). Quit.
Step 2: Divide $\boldsymbol{V}$ into two subsets $\boldsymbol{V}_L$ and $\boldsymbol{V}_R$ of approximately equal cardinalities so that $\boldsymbol{V}_L$={v_p, …, v_q} and $\boldsymbol{V}_R$={v_{q+1}, …, v_r}, where $q = (p + r) / 2$.
Step 3: If (j mod 2)= 0 then
$\boldsymbol{DT}(\boldsymbol{V})$ = *merge*(*delaunay2d*($\boldsymbol{V}_L$), *delaunay2d*($\boldsymbol{V}_R$), j+1)
else
$\boldsymbol{DT}(\boldsymbol{V})$ = *merge*(*delaunay2d*($\boldsymbol{V}_R$), *delaunay2d*($\boldsymbol{V}_L$), j+1).

The algorithm *delaunay2d* is not very different from the conventional D&C algorithms,[2,3] except for the fact that the merge

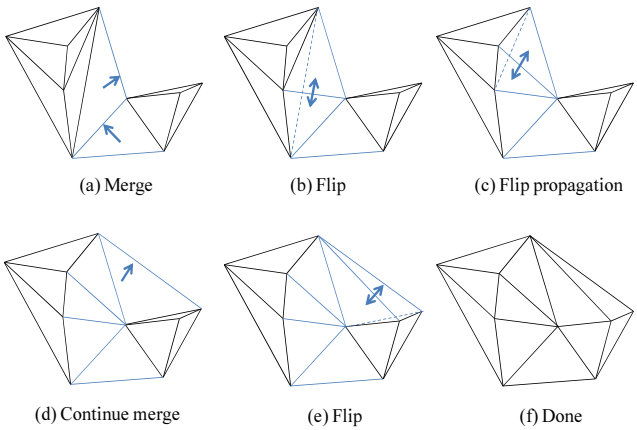


Fig. 5 Merging with *flip* operations

is accomplished in two-dimensional directions. In *Step* 1, if the number of elements of the vertex array $\boldsymbol{V}$ is less than or equal to three, no merge is necessary. Then *elementary-delaunay* creates a triangle if $|\boldsymbol{V}| = 3$ and the three vertices are not collinear. Otherwise, if $|\boldsymbol{V}| = 2$, *elementary-delaunay* leaves the region so that they can be merged in the upper level. In *Step* 2, the divided vertex arrays noted as $\boldsymbol{V}_L$ and $\boldsymbol{V}_R$ also represent the lower and upper regions in the two-dimensional space, as well as the left and right regions. Step 3 invokes the *merge* algorithm with the two divided vertex arrays along horizontal or vertical directions.

Triangulation is actually performed in the *merge* process that combines two sub-triangulations into a larger triangulation. The following algorithm assumes that $\boldsymbol{DT}(\boldsymbol{V}_L)$ and $\boldsymbol{DT}(\boldsymbol{V}_R)$ have at least one triangle:

Algorithm *merge*
input: $\boldsymbol{DT}(\boldsymbol{V}_L)$, $\boldsymbol{DT}(\boldsymbol{V}_R)$, where $\exists\, t_L \in \boldsymbol{DT}(\boldsymbol{V}_L)$ and $\exists\, t_R \in \boldsymbol{DT}(\boldsymbol{V}_R)$.
output: $\boldsymbol{DT}(\boldsymbol{V})$, where $\boldsymbol{V} = \boldsymbol{V}_L \cup \boldsymbol{V}_R$.

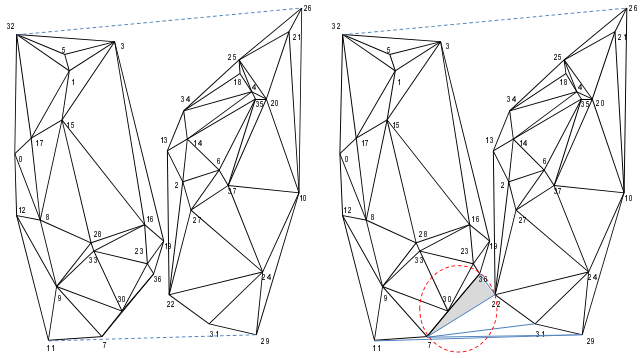
Find Convex Boundry
Step 1: Find edge $e_{base}(v_{Ls}, v_{Rs})$ and $e_{top}(v_{Le}, v_{Re})$ for $\boldsymbol{V}_L$ and $\boldsymbol{V}_R$.
Step 2: Set $t_{base} = \text{ø}$, $v_{L0} = v_{Ls}$, $v_{R0} = v_{Rs}$.
Step 3: If $edge(v_{L0}, v_{R0}) = e_{top}$ then quit.
Step 4: Set $t_L = cwt(v_{L0})$, $t_R = ccwt(v_{R0})$, $v_{L1} = tnexv(t_L, v_{L0})$, and $v_{R1} = tprev(t_R, v_{R0})$.
Step 5: If (v_{L1}, v_{L0}, v_{R0}) can be a triangle, then do *Step* 6~9 else do *Step* 10~13.
Step 6: New triangle $t = create\_neighbor(t_L, v_{R0}, index(t_L, v_{L0})+2)$.
Step 7: $flip\_p2(t, index(t, v_{R0}))$ and set $t = mccwt(v_{L1})$.
Step 8: If $t_{base} \neq \text{ø}$ then $mate(t, t_{base})$, $flip\_p4(t, index(t, t_{base}))$, and set $t = mccwt(v_{L1})$.
Step 9: Set $t_{base} = t$ and $v_{L0} = v_{L1}$. go to *Step* 3.
Step 10: New triangle $t = create\_neighbor(t_R, v_{L0}, index(t_R, v_{R0})+1)$.
Step 11: $flip\_p2(t, index(t, v_{L0}))$ and set $t = mcwt(v_{R1})$.
Step 12: If $t_{base} \neq \text{ø}$ then $mate(t, t_{base})$, $flip\_p4(t, index(t, t_{base}))$, and set $t = mcct(v_{R1})$.
Step 13: Set $t_{base} = t$ and $v_{R0} = v_{R1}$. go to *Step* 3.

The purpose of *Step* 1 is to find the convex boundary of the two sub-triangulations for a merge (as can be seen in Fig. 6(a)). The vertices on the upper and lower bounds for each sub-triangulation can be found using the permutation vector. These vertices form edges, called 'common tangents', and can be found in $O(\log n)$ using a boundary search of the *k*d-tree. The steps between *Step* 3 and *Step* 12 are repeated, while the base line changes from the lower common tangent to the upper common tangent. The operators $cwt(v, t)$ and $ccwt(v, t)$ retrieve the adjacent triangle t, which shares v and lies on the clockwise and counterclockwise sides of t. The operators $mcwt(v)$ and $mccwt(v)$ traverse adjacent triangles of the vertex v in clockwise and counterclockwise directions to find triangles on convex bound. The operator $create\_neighbor(t, v, i)$ creates a new triangle adjacent to the triangle t and a vertex v, which is located outside of t, and the new triangle can then be referred to by $tri(t, i)$. The operator *mate* ensures that the two adjacent triangles properly refer to each other.

It is important to note that *Step*s 6-9 and 10-13 are symmetric. One merges from a triangle on the left triangulation $\boldsymbol{DT}(\boldsymbol{V}_L)$, and the other on the right triangulation $\boldsymbol{DT}(\boldsymbol{V}_R)$. When a triangle has been created between the two triangulations, it is necessary to preserve the empty circle property for Delaunay triangulation. Figure 6(b) details the moment when a new triangle has been created which violates the empty circle property. The FBD&C algorithm preserves the empty circle property with the propagation of edge flipping, as in the previously reported incremental algorithm.[17] The operations $flip\_p2(t, i)$ and $flip\_p4(t, i)$ propagate flip operations unless $tri(t, i)$ satisfies the empty circle property. These flip propagations are discussed in detail in the next section.

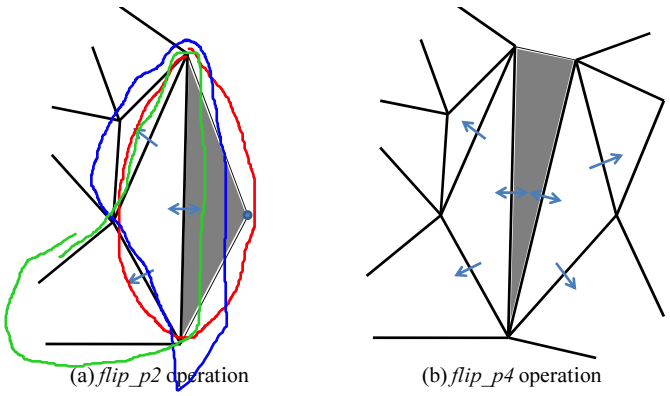
3.3.3 Flip propagation

There are two types of flip propagation in the FBD&C. The first one is two-directional, and the other is four-directional. Two-directional propagation is denoted as $flip\_p2(t, i)$, which is similar to the edge flip operation generally used in incremental constructions. Four-directional propagation, as proposed in this paper, is denoted as $flip\_p4(t, i)$. Four-directional flip propagation is not necessary in an incremental construction of Delaunay triangulation, but frequently occurs in the FBD&C. Figure 7 shows a comparison of *flip_p2* and *flip_p4*. If the shaded triangle in Fig. 7 (a) is created and does not satisfy the empty circle property, a flip occurs and is propagated to the other two edges. If a triangle is created between the two triangles, as shown in Fig. 7(b), then flip propagations simultaneously occur on the two adjacent edges and propagate in four directions.



(a) Finding convex boundary (b) Flip impending state

Fig. 6 Flip occurs in merging triangulations



(a) *flip_p2* operation (b) *flip_p4* operation

Fig. 7 Comparison between *flip_p2* and *flip_p4*

flip_p2 and flip_p4 are testing if the delaunay triangles are meeting the angle property between 2 triangles

The following algorithm *flip_p2*(t, i) tests and flips *edge*(t, i) if necessary. After the flip, it propagates to edges between *tri*(t, i) and two triangles other than t, until all of the visited triangles satisfy the empty circle property.

Algorithm *flip_p2* goes through a triangles neighbors
input: *edge*(t_0, i_0)
output: Modified triangulation ***DT***, where $t_0 \in \boldsymbol{DT}$
Step 1: If tri(t_0, i_0)=ø then quit.
Step 2: Set t_1 = *tri*(t_0, i_0) and i_1 = *index*(t_1, t_0).
Step 3: If *vert*(t_1, i_1) is not included by the *circumcircle* of t_0, quit.
Step 4: Do *flip*(t_0, i_0).
Step 5: Call *flip_p2*(*edge*(t_0, i_0)).
Step 6: Call *flip_p2*(*edge*(t_1, i_1+1)).

The algorithm starts with an edge and repeats while the propagated edge violates the empty circle property. If the edge is on the boundary of the triangulation, the empty circle test is not necessary (*Step* 1). The empty circle property is tested with the neighbor triangle in *Step* 2 and *Step* 3, and, if not satisfied, flipping occurs in *Step 4*. After that, the test and flipping should propagate to other edges. Notably, the indices change when the *flip* operation is applied; thus, the edges referenced by the changed indices are pushed to the queue. The edges *edge*(t_1, i_1+1) and *edge*(t_1, i_0+2) would be the next ones to be tested and propagated before the *flip* operation in *Step* 4; however, the topology is altered by the *flip* operation, as shown in Fig. 3, thus, the next two edges are *edge*(t_0, i_0) and *edge*(t_1, i_1+1).

The four-directional flip propagation *flip_p4* is more complicated than *flip_p2*. If a new triangle is created and is adjacent to only one triangle, as shown in Fig. 8(a), then *flip_p2* is applicable. However, if the new triangle t_{new} is created between the adjacent triangles t_{base} and t_R, then the flip should propagate from the two adjacent edges, as shown in Fig. 8(b). If the flips for the two edges are processed sequentially, then topology changes will occur, and then propagation will continue, as in Fig. 8(c). Keeping the propagation path from the two edges is more complicated than it would be from one edge, because the two propagation paths may overlap. Therefore, a four-directional edge flip propagation *flip_p4* was devised, as shown in Fig. 8(d), to define the propagation in a simpler manner. The flip operations start from an edge and propagate in four directions.

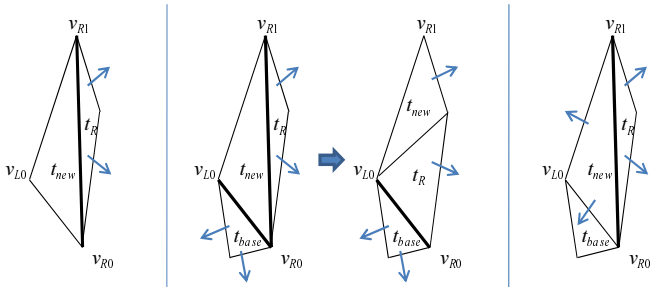


Fig. 8 Flip propagation in four directions

Algorithm *flip_p4*
input: *edge*(t_0, i_0), triangle geometry
{*vert*(t_0, 0), *vert*(t_0, 1), *vert*(t_0, 2)}
output: Modified triangulation ***DT***, where $t_0 \in \boldsymbol{DT}$
Step 1: If *tri*(t_0, i_0) = ø or $t_0 \neq (v_0, v_1, v_2)$ then quit.
Step 2: Set t_1 = *tri*(t_0, i_0) and i_1 = *index*(t_1, t_0).
Step 3: If *vert*(t_1, i_1) is not included by the *circumcircle* of t_0, then quit.
Step 4: Do *flip*(t_0, i_0).
Step 5: Set $t_a = (t_0, i_0)$, $t_b = (t_1, i_1+1)$, $t_c = (t_1, i_1)$, and $t_d = (t_0, i_0+1)$.
Step 6: Call *flip_p4*(*edge*(t_a, *index*(t_a, t_0)), {*vert*(t_a, 0), *vert*(t_a, 1), *vert*(t_a, 2)}).
Step 7: Call *flip_p4*(*edge*(t_b, *index* (t_b, t_1)), {*vert*(t_b, 0), *vert*(t_b, 1), *vert*(t_b, 2)}).
Step 8: Call *flip_p4*(*edge*(t_c, *index*(t_c, t_1)), {*vert*(t_c, 0), *vert*(t_c, 1), *vert*(t_c, 2)}).
Step 9: Call *flip_p4*(*edge*(t_d, *index*(t_d, t_0)), {*vert*(t_d, 0), *vert*(t_d, 1), *vert*(t_d, 2)}).

This algorithm is similar to *flip_p2*, with two major differences. The first is that *flip_p4* propagates to four-directions, and the second is that *flip_p4* requires the triangle geometry information {*vert*(t_0, 0), *vert*(t_0, 1), *vert*(t_0, 2)}. The geometry of a triangle can be described with three indices of the vertices *vert*(t_0, 0), *vert*(t_0, 1), and *vert*(t_0, 2) in the vertex array. This triangle geometry is used for checking the availability in the case of multiple visits to the triangle. If an edge *edge*(t, i) is retrieved, but t is not available, we recognize that the edge has already flipped and, thus, determine that it is necessary to stop the current thread of the propagation. The indices in *Step* 6 were based on the changed topology after the *flip* operation in *Step* 4. The subsequent edges and triangle geometries used in the recursive propagations are shown in *Step* 6~9.

4. Experimental results and discussions

The time consumption $T(n)$ for the entire process of FBD&C with n sites is:

$$T(n) = T_{tree}(n) + T_{dc}(n).$$

The total time required is the sum of time $T_{tree}(n)$ for space partitioning and time $T_{dc}(n)$, for performing the D&C triangulation. Although each term is divided in half for each level of recursion, time $T_{dc}(n)$ is more significant because the process is more complicated. Since $T_{tree}(n)$ is $O(n\log n)$, which is asymptotically optimal,[16] only $T_{dc}(n)$ is discussed in this section.

Lemaire showed that the D&C algorithm with the kd-tree is bounded to $\Omega(n\log n)$ for quasi-uniformly distributed sites on two-dimensional spaces,[8] asserting that the average time complexity is $O(n)$, with reference to Dwyer and, Guibas and Stolfi.[3,6] Dwyer identified three elements that significantly affect the D&C triangulation time: the number of sites visited to calculate the convex hull, the number of edges deleted, and the number of edges created. However, in our new approach, there is no edge deletion

process to consider. Furthermore, the balanced permutation vector used in the FBD&C helps to efficiently search the convex boundary. Therefore, the number of flip operations is the most significant factor in computing time.

The D&C triangulation time $T_{dc}(n)$ can be represented as:

$$T_{dc}(n) = 2T_{dc}(n/2) + T_{merge}(n), \text{ where } T_{merge}(n) = 2T_{cvx}(n/2) + T_f(n).$$

Where $T_{cvx}(n)$, and $T_f(n)$ are the times required for finding common tangents and flip propagations respectively. $T_{cvx}(n)$ is the same as the time required for the boundary search in the kd-tree; thus, it is denoted as $c_1 \log n$ ($c_1 > 0$). $T_f(n)$, which includes stitch and flip time, varies with the distribution of the sites, and is, thus, not easy to predict theoretically. Therefore, experiments were performed in order to investigate characteristics of the flip occurrences. Figure 9 shows an example of a merging of two sub triangulations, $\boldsymbol{DT}_L(n/2)$ and $\boldsymbol{DT}_R(n/2)$, into the resulting triangulation $\boldsymbol{DT}(n)$ for sites with quasi-uniform distribution.

Figure 10 presents the experimental results that show the number of flip occurrences during a merge for quasi-uniformly distributed sites (See Fig. 9). The number of flip occurrence oscillates, but it has a tendency to be roughly linear. Note that the coefficient of the linear regression is 0.062, which indicates that the flip occurrences modestly increase with the number of sites in a merge.

Katajainen and Koppinen,[8] and, Lemaire and Moreau[9] selected sampled points from a logarithmic spiral curve $\rho=e^{\theta}$, $(-\pi/4 \le \theta \le \pi/4)$, to represent worst-case data, because inserting a new site may cause the destruction of all of the already constructed Delaunay triangles (See Fig. 11). The proposed algorithm was tested with a point set sampled from the same logarithmic spiral curve.

Figure 12 shows the flip occurrences during the merge process with the points sampled from the logarithmic spiral curve. The tendency towards flip occurrences is similar to that seen in Fig. 10, but the coefficient of the linear regression in this case was larger than that of the quasi-uniform distribution case. However, for both of the two types of site distribution - the points with quasi-uniform distribution and the points on the logarithmic spiral - the occurrence of the flips linearly increased with the number of sites.

The triangulation time of the FBD&C computation for various numbers of randomly generated sites was measured. The test

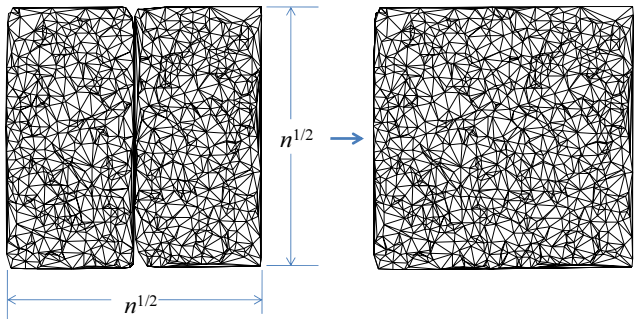


Fig. 9 Merging triangulations with quasi-uniform distribution of sites

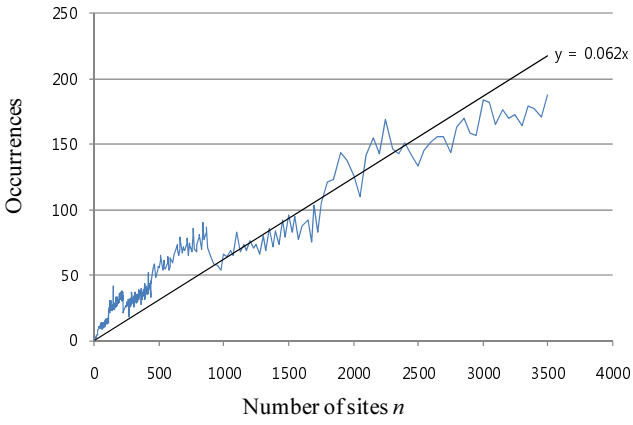


Fig. 10 Flip occurrences (quasi-uniform distribution of sites)

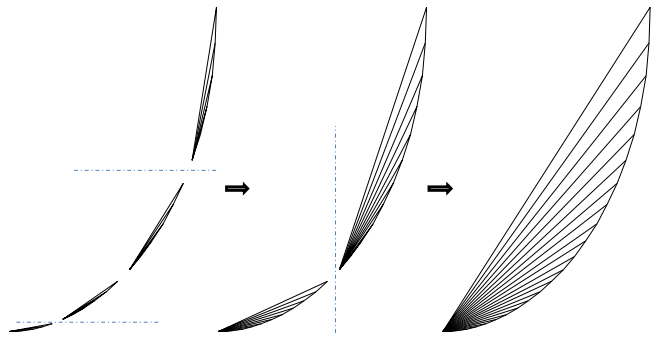


Fig. 11 Merging triangulations of sites on logarithmic spiral curve

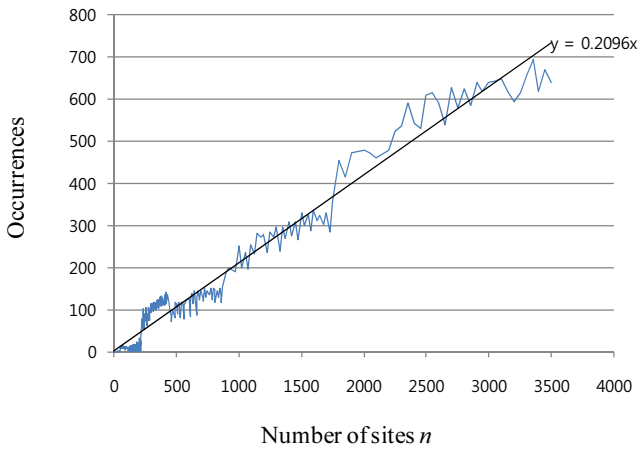


Fig. 12 Flip occurrences (sites on a logarithmic spiral curve)

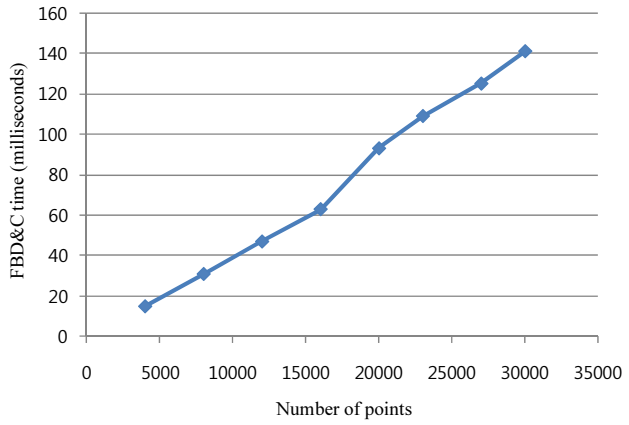


Fig. 13 Running time for FBD&C on 2.4GHz CPU

program was written in the C++ programming language. Since the running time depends on the number of flip occurrences, and flips are affected by site distribution, a fixed random seed was used to achieve identical site distribution. The test results shown in Fig. 13 were obtained on a Windows XP desktop PC with 2.4 GHz Intel Core2 Quad CPU and 3 GB RAM.

The test results in Fig. 13 show that the time approximately linearly increased with the number of sites n. The calculation times were 93 milliseconds for 20,000 points and 141 milliseconds for 30,000 points. As we observed in the experiments of counting flip occurrences for the two different types of data sets, $T_f(n)$ is estimated to be linear to n. Therefore, the proposed triangulation algorithm is an expected $O(n\log n)$ algorithm. However, the test results shown in Fig. 13 show empirically linear time complexity for the quasi-uniformly distributed sites set.

5. Conclusions

A new D&C Delaunay triangulation algorithm is proposed in this paper. The merge process of the D&C triangulation algorithm was simplified by skipping the pre in-circle test found in conventional D&C algorithms. The empty circle property of the triangulation is still preserved with a flip propagation algorithm for the merge process. Although it is difficult to estimate the number of flip occurrences in a theoretical way, the experimental results show that the flip propagations do not occur excessively; therefore, the algorithm remains $O(n\log n)$. The proposed algorithm can be implemented in an easy and concise manner, since there is no deletion of the existing edges. Thus, the bookkeeping of complex intermediate topological states is not necessary. A simple array-based triangle data structure was effectively used for implementation, rather than the commonly used edge-based structures. The use of an array-based triangle data structure was possible because it was not necessary to trace edge deletion and creation in the merge process. The space partitioning information is represented as a permutation vector sequence in a vertices array; therefore, no additional data is required for space partitioning.

ACKNOWLEDGEMENT

This research was supported by the Chung-Ang University Research Scholarship Grants in 2009 and 2010.

REFERENCES

1. Shamos, M. I. and Hoey, D., "Closest-point problems," Proc. of 16th Annual IEEE Symp. on FOCS 1975, pp. 151-162, 1975.
2. Lee, D. T. and Schachter, B. J., "Two algorithms for constructing a Delaunay triangulation," Int. J. Comput. Inf. Sc., Vol. 9, No. 3, pp. 219-242, 1980.
3. Guibas, L. and Stolfi, J., "Primitives for the manipulation of general subdivisions and the computation of Voronoi Diagrams," ACM Transactions on Graphics, Vol. 4, No. 2, pp. 74-123, 1985.
4. Ohya, T., Iri, M. and Murota, K., "A fast Voronoi-diagram algorithm with quaternary tree bucketing," Information Processing Letters, Vol. 18, No. 4, pp. 227-231, 1984.
5. Maus, A., "Delaunay Triangulations and the convex hull of n points in expected linear time," BIT Numerical Mathematics, Vol. 24, No. 2, pp. 151-163, 1984.
6. Dwyer, R. A., "A faster divide-and-conquer algorithm for constructing Delaunay triangulations," Algorithmica, Vol. 2, No. 1-4, pp. 137-151, 1987.
7. Katajainen, J. and Koppinen, M., "Constructing Delaunay triangulations by merging buckets in quadtree order," Fundamenta Informaticae, Vol. 11, No. 3, pp. 275-288, 1988.
8. Lemaire, C. and Moreau, J. M., "A probabilistic result on multi-dimensional Delaunay triangulations and its application to the 2d case," Computational Geometry, Vol. 17, No. 1-2, pp. 69-96, 2000.
9. Bentley, J. L., "Multidimensional binary search tree in database applications," IEEE Transactions on Software Engineering, Vol. 5, No. 4, pp. 333-340, 1979.
10. Moura, A. L., Camacho, J. R., Guimaraes Jr., S. C. and Salerno, C. H., "A functional language to implement the divide-and-conquer Delaunay triangulation algorithm," Applied Mathematics and Computation, Vol. 168, No. 1, pp. 178-191, 2005.
11. Drysdale, R. L., "A practical algorithm for computing the Delaunay triangulation for convex distance functions," Proc. of Symp. on Discrete Algorithms archive, pp. 159-168, 1990.
12. Leach, G., "Improving worst-case optimal Delaunay triangulation algorithms," Proc. of 4th Canadian Conference on Computational Geometry, pp. 340-346, 1992.
13. Cignoni, P., Montani, C. and Scopigno, R., "DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d," Computer-Aided Design, Vol. 30, No. 5, pp. 333-341, 1998.
14. Yang, S. W., Choi, Y. and Lee, H. C., "CAD data visualization on mobile devices using sequential constrained Delaunay triangulation," Computer-Aided Design, Vol. 41, No. 5, pp. 375-384, 2009.
15. Burkhard, W. A. and Keller, R. M., "Some approaches to best match file searching," Communication of ACM, Vol. 16, No. 4, pp. 230-236, 1973.
16. Bentley, J. L., "Multidimensional binary search trees used for associative searching," Communications of ACM, Vol. 18, No. 9, pp. 509-517, 1975.
17. Anglada, M. V., "An improved incremental algorithm for constructing restricted Delaunay triangulations," Computers & Graphics, Vol. 21, No. 2, pp. 215-223, 1997.