Visualizing and representing the evolution of topological features
Fonseca, Rasmus; Jørgensen, Desirée Malene Schreyer

Publication date: 2012

Document Version
Early version, also known as pre-print

Citation for published version (APA):
Visualizing and representing the evolution of topological features

Rasmus Fonseca* Desirée Malene Schreyer Jørgensen†

Abstract

Simplicial complexes are discrete representations of topological spaces that are practical for computational studies. The first three Betti-numbers (indicating the number of components, tunnels and voids), as well as the topological persistence of each such feature, is well-defined and can be efficiently computed for simplicial complexes embedded in 2D and 3D [1, 2].

We introduce a novel representation of the evolution of topological features in simplicial complexes using so-called tunnel-trees in 2D and void-trees in 3D. This new representation makes it possible to analyze topological evolution by applying tools for analysis of binary trees. Furthermore it supplies a new method for visualizing topological evolution.

Introduction

A simplicial complex, $K$, is a set of simplices where any face of a simplex in $K$ is also in $K$ and the intersection of two simplices in $K$ is either empty or a face of both simplices. Delfinado and Edelsbrunner [1] define a filter to be a sequence of simplices, $\sigma^1, \sigma^2, \ldots, \sigma^n$, where $i = \{\sigma^1, \sigma^2, \ldots, \sigma^n\}$ is a simplicial complex for any choice of $i$ (see left part of Figure 1). The filter represents the evolution of a simplicial complex and will be the focus of the methods described here. The topological features of a complex can be described using the Betti-numbers, $\beta_d$, which indicate the rank of the $d$th homology group. The first three Betti-numbers ($\beta_0, \beta_1, \beta_2$) can be interpreted more intuitively as the number of components, holes, and voids respectively. A $O(na(n))$-time algorithm exists to calculate the evolution of $\beta_d$ as a simplicial complex is grown using a filter [1]. This method identifies each $k$-simplex, $\sigma^i$, as either positive if it creates a new $k$-cycle and thereby increases $\beta_k$, or negative if it changes a $k$-cycle into a $k$-boundary and thereby decreases $\beta_{k-1}$. For each positive $k$-simplex, $\sigma^i$, the negative $(k+1)$-simplex, $\sigma^j$, that is responsible for turning the $k$-cycle, created by $\sigma^i$, into a $k$-boundary can be efficiently identified [2]. The difference between the indices of such two simplices is defined to be the persistence of the $k$-cycle represented by $\sigma^i$.

Tunnel- and void-trees

One interesting observation about tunnels in simplicial complexes embedded in 2D is that, often, when a positive 1-simplex (edge) is added to the complex, it splits one tunnel in two. If the empty space around the complex is considered a bounding tunnel, then every positive edge will split an existing tunnel in two. Similarly, if the entire space around a simplicial complex embedded in 3D is considered a bounding void, then a positive 2-simplex (triangle) always splits an existing void in two.

Based on this observation we define a tunnel-tree (or $\beta_1$-tree) of a 2D filter to be a binary tree where each node represents a distinct tunnel (see right part of Figure 1). The root is the bounding tunnel, and the leaves are triangular tunnels that will not be split further. With each node $n$ we associate the positive edge that represents the tunnel, $\epsilon(n)$, and with each leaf, we associate the negative triangle that fills this tunnel, $\tau(n)$. The tunnel-tree is ordered such that for any node $n$, the triangle of the rightmost leaf, $\tau($Tree-Max$(n))$, is the triangle that ‘destroys’ $\epsilon(n)$ and hence determines its persistence. A void-tree (or $\beta_2$-tree) of a 3D filter is defined in a similar fashion, only with positive triangles as nodes and negative tetrahedra as leaves.

$A \beta_k$-tree is constructed by running through the filter backwards as shown in Algorithm 1. Leaves are created when a negative $(k+1)$-simplex is encountered and the roots of leaves are connected when positive $k$-simplices are encountered.

Algorithm 1 Build a $\beta_k$-tree given a filter

1: Create a 'bounding node', $n_b$
2: for $i = n$ to 1 do
3: if $\sigma^i$ is a negative $(k + 1)$-simplex then
4: Create a new node, $n$, and set $\tau(n) \leftarrow \sigma^i$
5: else if $\sigma^i$ is a positive $k$-simplex then
6: $(n_0, n_1) \leftarrow$ Nodes of the two $(k + 1)$-simplices adjacent to $\sigma^i$
7: $(n_0, n_1) \leftarrow \left(\text{ROOT}(n_0), \text{ROOT}(n_1)\right)$
8: Swap $n_0$ and $n_1$ if $\tau($Tree-Max$(n_0))$ is younger than $\tau($Tree-Max$(n_1))$
9: Create a new node $n$ with $n.left \leftarrow n_0$, $n.right \leftarrow n_1$, and $\epsilon(n.left) \leftarrow \sigma^i$
10: end if
11: end for
12: return Root($n_b$)

---

*Department of Computer Science, University of Copenhagen, rfonseca@diku.dk
†Department of Computer Science, University of Copenhagen, daisy@diku.dk
In line 4, the \((k + 1)\)-simplex can be associated with its node using a hash-map. This ensures that locating the nodes of adjacent \((k + 1)\)-simplices in line 6 can be performed in constant time. In line 6, if one of the \((k + 1)\)-simplices adjacent to \(\sigma^i\) is not defined then the bounding node \(n_b\) is used instead. If \(\sigma^i\) has no adjacent \((k + 1)\)-simplices then a new node is created for \(n_0\), and \(n_1\) is set to \(n_b\). Line 8 guarantees that the youngest simplex in a subtree can always be found by going to the far right in the tree using Tree-Max.

A \(\beta_k\)-tree may be arbitrarily unbalanced, so a straightforward implementation will run in \(O(n^2)\) time worst case. The Tree-Max method can be improved to \(O(1)\) time by maintaining the maximum of each subtree as they are constructed. A data structure similar to disjoint-sets can be used to make the root method run in \(O(\alpha(n))\)-time, so the entire method runs in \(O(n\alpha(n))\) worst case time.

**Applications**

One attractive property of \(\beta_k\)-trees is that they give an alternative representation of the topological evolution of a filter. This can be used in several ways.

First, the fact that simplices in the subtree of a particular node will tend to be spatially close to each other gives rise to a new definition of local persistence. A particular edge, representing a tunnel, might be deemed particularly persistent if its subtree contains more than a certain number of nodes. Such a definition of persistence will not be affected by the addition of simplices outside the tunnel.

Using a Delaunay complex and the radius of the smallest empty circumcircle to generate an \(\alpha\)-filter [3], the arrangement of a particular sub-tree also gives an indication of the shape of the corresponding feature. For instance, a node with an unbalanced sub-tree indicates a tunnel that is narrowing, whereas a balanced node indicates a constant width.

For some applications, a tree might be a better visualization of the topological evolution than e.g. \(k\)-triangles [2]. The above mentioned properties of locality can be computationally analyzed, but they can also be derived simply by inspecting \(\beta_k\)-trees. The length of edges in the tree can furthermore be scaled to reflect the difference in birth time of the \(\epsilon(n)\) simplices.

Another interesting property of \(\beta_k\)-trees is that all \((k+1)\)-simplices within a particular tunnel/void are easily identified by locating the node in the tree with the desired \(\epsilon(n)\) and then collecting all leaves in the subtree using any tree-traversal method. In this manner the area of tunnels/volume of voids, for instance, is easily calculated.

Finally, any analysis method that works on trees is now applicable to topological evolutions. For instance the topology of two point-sets can be compared by finding the tree-edit-distance between the tunnel-trees (or void-trees) of their respective \(\alpha\)-filters.

**References**

