Lecture 16: Computation of Reeb Graphs

Topics in Computational Topology: An Algorithmic View

Scribed by: Jack Cheng

1 Introduction

Given a manifold X and a function $f : X \to \mathbb{R}$, the level set of X containing the point q is the set $\{x \in X : f(x) = f(q)\}$. Each connected component of the level set is called a *contour*. Recall from last lecture that the *Reeb graph* $R_f(X)$ of X on f is defined to be the continuous contraction of each contour to a point. Recall furthermore that if the manifold is simply-connected (i.e., all loops are contractible to a point), then the Reeb graph has a tree structure, and the resulting Reeb graph is called a *contour tree*.

In this lecture, we first discuss an efficient algorithm for computing contour trees. After that, a survey of various algorithms for computing Reeb graphs will be given.

1.1 Assumptions

As before, we assume X is a simplicial complex and f is a nice piecewise-linear function defined on the vertices of X. Since the connectivity of each level set can be described completely by the 2-skeleton of X, only simplices of dimension 2 or less are needed. We refer to those simplices by their common names: vertices, edges, triangles. The generic assumption that each vertex has distinct function values is also imposed.

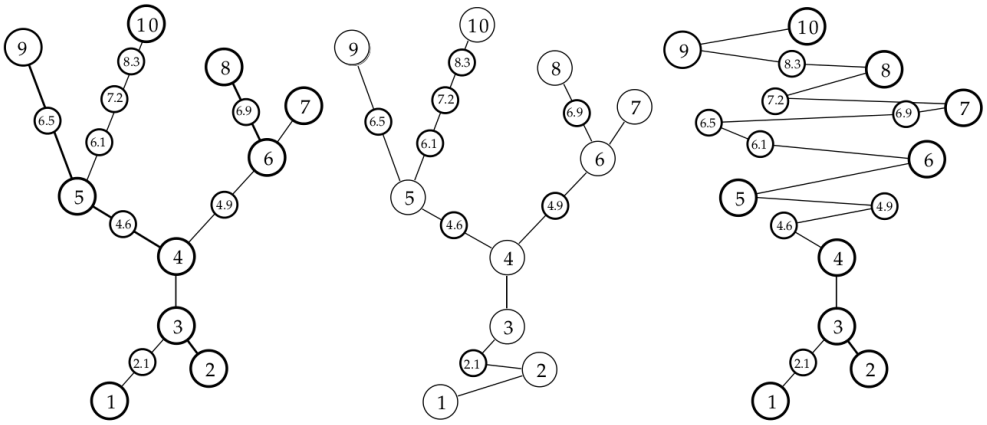
2 Computation of Contour Trees

The popular algorithm for computing contour trees was introduced in [1]. The algorithm works by computing a join tree and a split tree of X, then novelly combining those trees to form a contour tree. The algorithm runs in $O(n \lg n + N + t\alpha(t))$, where n is the number of vertices, N is the number of edges, t is the number of supernodes in the contour tree, and α is the slow-growing Inverse Ackerman function. Note that only the 1-skeleton is needed to find the contour tree.

2.1 Join Tree and Split Tree

Suppose X is swept in increasing order of function value, then the join tree encodes all occurrence in X when two components are merged. The split tree analogously encodes all occurrences when a component is split into multiple components. See Figure 1. Observe that the split tree is a join tree with f negated (i.e., sweeping in decreasing order of function value instead of increasing order). Hence, it suffices to give an algorithm for computing the join tree. The split tree can be computed using the same algorithm with $f' = -f$.

The join tree can be computed easily using a union-find data structure, augmented with an extra field that encodes the highest vertex in a component. Specifically, the algorithm traverses the vertices of the complex in increasing order of function value. At each vertex v_i, a new component is created, with the highest vertex being v_i. Then all adjacent vertices v_j with a lower function value are considered and processed as follows:



(a) Augmented Contour Tree (b) Join Tree (c) Split Tree

Figure 1: The join and split tree of an augmented contour tree. The larger nodes are the actual nodes of the contour tree. Note that f increases from the top of the figure to the bottom. Image taken from [1]

if `find(`v_i`)` $\neq$ `find(`v_j`)` **then**
  `union(`v_i`,` v_j`)`
  Add edge between v_i and `find(`v_j`).highestVertex` to the join tree.
end if

The processing for v_i ends with updating `find(`v_i`).highestVertex` to v_i.

2.2 Merging the Join Tree with the Split Tree

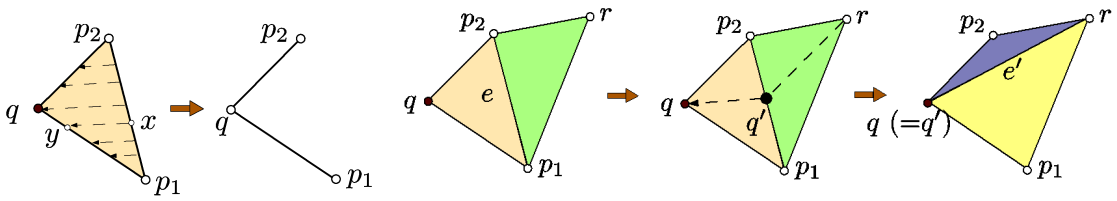
The novelty of the algorithm lies in how the join tree and split tree are merged to form the contour tree. The intuition behind the merging is the following: a leaf vertex in either the join tree or the split tree that is also a degree-2-or-less vertex in the other tree can be removed and added to the contour tree easily. In particular, say v_i is a leaf node in the join tree and a degree-2-or-less vertex in the split tree. Then the edge between v_i and its adjacent vertex in the join tree must also be present in the contour tree. Moreover, v_i can be removed from the join and split trees by simply removing the vertex and its edges. In the case of a degree-2 vertex, the two adjacent vertices of v_i are connected by edge afterward to maintain the tree's connectivity.

The algorithm repeats this process of picking such a leaf vertex v_i, adding its edges to the contour tree, then removing it from the join and split trees. Since there are no loops, this process will exhaust all the vertices, allowing the contour tree to thus be constructed inductively. For more details, as well as rigorous proofs, see the original paper.

3 Computation of Reeb Graphs

3.1 Sweep-based Algorithms

The first provably-correct algorithm for computing Reeb graphs is attributed to Shinagawa and Kunii [6]. The algorithm works by sweeping the vertices of the complex in increasing order of function value. At each vertex, it keeps track of the triangles intersecting the current level set, allowing the connected components to be computed. The level sets are updated by replacing the triangles in the current vertex's lower star with the triangles in the upper star. The original paper only dealt with 2-manifolds, with a quadratic running



(a) Collapse (b) Split

Figure 2: Illustration of a triangle's collapse and split operations.

time. However, the same idea generalizes to arbitrary simplicial complexes with a running time of $O(nm)$, where n is the number of vertices and m is the size of the 2-skeleton (i.e., the number of vertices, edges, and triangles).

As a quick remark, note that the union-find data structure by itself cannot keep track of the level sets, since it does not support the operation of removing a triangle.

By taking advantage of the simpler nature of level sets in a 2-manifold (i.e., loops), [2] improved the running time for 2-manifolds to $O(m \lg n)$ by using a balanced binary tree to maintain and update the level set. Using more complex search trees, [3] improved the running time for d-manifolds to $O(m \lg m (\lg \lg m)^3)$.

3.2 Other Algorithms

Recently, [7] proposed an algorithm for 3-manifolds with boundary embedded in $\mathbb{R}^3$ with an expected running time $O(m \lg m + hm)$, where h is the number of independent loops in the Reeb graph. The algorithm works by transforming the input manifold into a loop-free one, allowing efficient contour tree algorithms to be used. The worst-case time is $O(nm)$, although it runs much faster in practice, since h is usually small.

An online algorithm was proposed in [5] that worked for arbitrary simplicial complexes. The algorithm is simple and does not require the simplices to be processed in a specific order. It works by merging the edges from each triangle as the triangles are encountered. By aggressively removing vertices and edges that are completely processed, the algorithm works very efficiently and requires little space. Its worst-case timing complexity, however, is still $O(nm)$.

3.3 Randomized $O(m \lg n)$ Algorithm

This is first sub-quadratic algorithm for computing Reeb graphs (although in the average case only). It takes a different approach than the previous algorithms: instead of considering the evolution of the contours, this algorithm focuses on the surjection from X to $R_f(X)$. It systematically collapses each triangle in X until only edges remain. By making sure that each collapse operation preserves the Reeb graph, it is easy to see that the resulting graph, which consists of vertices and edges only, must be a Reeb graph. Before we describe the algorithm, some notations are needed:

Definition 3.1 *Given a triangle t, the* mid vertex *is the vertex with the intermediate function value of the three vertices. The* long edge *of t is the edge on the opposite side of mid vertex (i.e., it is the edge that spans the entire range of function values on t).*

Definition 3.2 *A horizontal segment of t containing x is the set of all points $q \in t$ such that $f(q) = f(x)$.*

In the algorithm, a triangle t is collapsed as follows: let e_1, e_2 be the edges adjacent to the mid vertex of t. Then for all points $x \in e_1 \cup e_2$, we collapse the horizontal segment containing x to x. (See Figure 2a.)

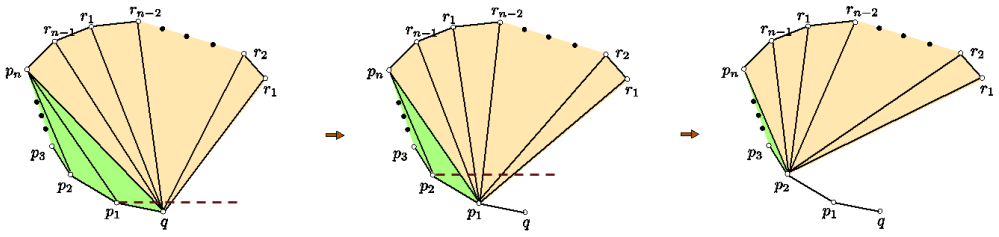


Figure 3: Example of when the non-randomized algorithm can take $O(nm)$ time.

If the long edge e_3 of t has neighboring triangles, the resulting complex will not be simplicial. To make it an abstract simplicial complex, a *split* operation is done for each neighboring triangle s. This operation creates an edge from the mid vertex of t to the vertex of s opposite e_3 (see Figure 2b). Note that "abstract" is needed here, since the two resulting simplices might intersect.

For each vertex q, the collapse operation can be performed until there are no more triangles with q as the mid vertex. To see that this terminates, observe that each collapse operation reduces the number of triangles intersecting the contour containing q by one. Since that number is always nonnegative, the process must stop. As a consequence, after this process ends for q, there are no more triangles intersecting the contour containing q. We term this sequence of collapse operations the *vertex-collapse* of q.

An algorithm to compute the Reeb graph could simply be to perform the vertex-collapse operation on all the vertices. However, as it is, this algorithm is still $O(nm)$. This is because the vertices could be processed in an order where each is the mid vertex of $\Theta(n)$ triangles. As an example, see to Figure 3.

The example also illustrates the problem with above algorithm. Observe that when the vertex p_1 is collapsed, each new split triangle still intersects the contour containing p_i for all $i > 1$. The same is true when p_2 is processed next, similarly for p_3, p_4, and so on. Hence, each of those split triangles have to be collapsed again and again for each p_i. Now, instead of processing the p_i in order, suppose $p_{n/2}$ is processed first, then $p_{n/4}$, then $p_{3n/4}$, and so on (i.e., we keep taking the midpoints). Then each new split triangle intersects only half of the original contours, meaning that each only need to be collapsed a logarithmic number of times.

Of course, we cannot really choose the optimal order every time. What we can do, however, is randomly choose the vertex to collapse. Then with constant probability, a vertex near the middle will be picked, giving us the desired $O(\lg n)$ bound. The complete algorithm is the following:

Randomly permute the vertices $V = \{v_1, v_2, \ldots, v_n\}$.
for all $v_i \in V$ **do**
  `vertex-collapse(`v_i`)`
end for

Note that the algorithm can be made more efficient by considering only the critical points of f. For more information, refer to the original paper [4].

References

[1] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry: Theory and Applications*, 24:75–94, 2003.

[2] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in reeb graphs of 2-manifolds. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, pages 344–350. ACM Press, 2003.

[3] H. Doraiswamy and V. Natarajan. Efficient algorithms for computing reeb graphs. *Computational Geometry: Theory and Applications*, 42:606–616, 2009.

[4] W. Harvey, Y. Wang, and R. Wenger. A randomized $o(mlogm)$ time algorithm for computing reeb graphs of arbitrary simplicial complexes.

[5] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26(3):58, 2007.

[6] Y. Shinagawa and T. Kunii. Constructing a reeb graph automatically from cross sections. *IEEE Computer Graphics and Applications*, 11:44–51, 1991.

[7] J. Tierny, A. Gyulassy, E. Simon, and P. V. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1177–1184, 2009.