Maintaining Persistence and Contour Trees for Time Varying Functions on 2 or 3-Manifolds

Issam Safa Yusu Wang*

Abstract

Topological methods have been gaining in prominence in the studies of scalar functions over a given domain. With the proliferation of time-varying functions in scientific data, the question of maintaining the data structures associated with topological methods over time becomes more and more important. In particular, suppose we are given a time-varying scalar function f defined over a manifold M, and a topological data structure associated with f. As f varies over time, the question is how to maintain the associated topological data structure such that it is always consistent with the scalar function f at any given point in time, without recomputing the data structure for f at each time instant.

In this paper, we consider two topological structures: topological persistence [16] and contour trees [26]. We present two algorithms: The first one maintains the persistence pairings for a time-varying function defined over a triangulation of a 2-manifold in $O(\log n)$ time per crossing event, where at a crossing event, two vertices switch the order of their function values. The second one maintains the contour tree of a time-varying function defined over a triangulation of a topological 3-sphere in $O(\mid lnk(p) \cap lnk(q) \mid \log n)$ time per crossing event, where $lnk(p)$ is the link of a vertex p in the 3-manifold. The second algorithm can also be extended to time-varying functions on d-manifolds with a slightly worse running time. It turns out that both these topological structures make use of a tree-like structure, and as such their time-varying versions can be solved efficiently with the help of the well-known dynamic tree data structure [23].

1 Introduction

Topological methods are useful tools in the study of scalar fields over manifolds. They cover a wide range of applications, from shape study to data denoising to efficient isocontouring to name few [7, 19, 22]. At the same time, we are witnessing a large proliferation of time-varying scalar functions across many application fields [3, 20, 25], and an investigation of topological methods in the context of time-varying data becomes a worthy endeavor given the ubiquity of such data. In particular, a natural question of interest is how to maintain the topological data structures at any moment in time, while avoiding a recomputation of those structures at each time instant. In this paper we consider two topological structures in the time-varying context: persistence pairings of critical points induced by a time-varying function over a triangulation of a 2-manifold, and contour trees for a time-varying function defined over a triangulation of the 3-sphere $\mathbb{S}^3$.

Persistent homology, first introduced in [16], is an algebraic method that can describe topological features induced by a filtered space. It provides a multi-scale way to measure the importance (persistence) of topological features. It has been recently applied to several shape analysis problems including feature identification, shape reconstruction, matching, and de-noising [4, 7, 9, 19]. In this paper, we consider the case where the input topological space is a triangulation K of some 2-manifold, and the persistence homology is induced by the sub-level sets of a piecewise-linear function f defined on K. This simple case still has many practical applications as scalar functions on surfaces present some of the most common scenarios in

*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA.

fields such as computer graphics and visualization. The standard persistence algorithm takes $O(n^3)$ time to compute all persistence pairings induced by a function f [17], where n is the size of K. However, for this simple case where K is a triangulation of a 2-manifold, it is well-known that the persistence pairings can be computed in $O(n\alpha(n))$ time (if vertices are already sorted) using the union-find data structure. In [6], Edelsbrunner et al. showed that in the general case, one can update persistence pairing in $O(n)$ time per *crossing event*, where a crossing event happens when two vertices switch the order of their function values. Since persistence pairing for static functions on 2-manifolds can be computed much more efficiently than the general case (i.e, $O(n\alpha(n))$ versus $O(n^3)$ time), we ask whether one can update the persistence pairings more efficiently for *time-varying* functions on 2*-manifolds* as well. We present a positive answer to this question in this paper, and show that the persistence pairings can be maintained in $O(\log n)$ time per crossing event in this case.

Contour tree is a topological structure that can track the evolution of the connected components in the level-sets of a scalar function over a simply connected domain. It has been used for seed selection for efficient isocontouring, and for segmenting and rendering individual components among many applications [1, 26, 27]. In [5], Carr et al. presented an elegant algorithm to compute the contour tree of a simply-connected domain K in any dimensions in $O(n \log n + N)$ time where n is the number of 0-simplices and N is the number of 1-simplices in K. Mascarenhas et al. showed that the contour tree can be updated in $O(n)$ time for each change in the tree that involves two critical points for a time-varying function on a triangulation of $\mathbb{S}^3$. In their approach, they used the Jacobi set, introduced in [11], to help identify necessary updates in the resulting contour tree. We consider a different approach in this paper using dynamic trees. Let $Lnk(p)$ denote the link of a vertex $p \in K$. Our algorithm maintains the contour tree for a time-varying function over a triangulation K of a simply connected 3-manifold in $O(\mid lnk(p) \cap lnk(q) \mid \log n)$ time for a crossing event between the endpoints of an edge in K, and in $O(\log n)$ time for all other crossing events. Note that $\mid lnk(p) \cap lnk(q) \mid$ can be $\Theta(n)$ in worst case. However, it is usually much smaller (in fact, often a constant) in practice. We remark that our approach processes every crossing event, while the algorithm from [14] processes a crossing event *only if* the two vertices involved are critical either before or after the crossing event.

Finally, we observe that the computation of both persistence pairings and contour tree result in a tree structure: the former due to the use of union-find procedure, and the latter itself is an unrooted tree. Studying the evolution of these topological quantities in a time-varying context thus amounts to studying the changes in those tree structures as the underlying function varies over time. Hence the high level framework for our approach for maintaining both persistence and contour tree is as follows: First, by a case-analysis of different crossing events, we analyze the types of updates that the associated tree structures may undergo. Next, by describing how to identify and perform these updates efficiently using dynamic trees [23].

Outline. In Section 2 we describe dynamic trees briefly, and introduce a set of operations and queries, implementable using dynamic tree operations, that will be used later in our algorithms. Section 3 gives some necessary notations. In Section 4 we describe our time-varying persistence-pairing algorithm for 2-manifolds. In Section 5, we present algorithm for maintaining time-varying contour trees for 3-manifolds.

2 Dynamic Trees

Dynamic trees, first introduced by Tarjan and Sleator [23], can maintain a collection of node-disjoint rooted trees that change over time as edges are added or deleted. The input tree is an acyclic graph with a real-valued cost associated with each node [24] (it can also work for cost associated with edges [23]). The following tree operations are defined for dynamic trees:

- $maketree(v)$: Create a new path containing the single node v, previously on no path, with cost zero.

- $findroot(v)$: Return the root of the tree containing node v.
- $findcost(v)$: Return $[w, x]$ where x is the minimum cost of a node on the tree path from v to findroot(v) and w is the last node (closest to the root) on this path of cost x.
- $addcost(v, x)$: Add real number x to the cost of every node on the tree path from v to $findroot(v)$.
- $link(v, w)$: Combine the trees containing nodes v and w by adding the edge $\langle v, w \rangle$. This operation assumes that v and w are in different trees and w is a root.
- $cut(v)$: Divide the tree containing node v into two or more trees by deleting the edge out of v. This operation assumes that v is not a tree root.

When the structure of the trees are represented implicitly, all these tree operations can be performed in $O(\log n)$ time. We will see later that all the update operations needed in this paper use a constant number of tree operations and thus run in $O(\log n)$ time each.

Finally, there is also an $evert(v)$ operation to re-root the tree at node v. This operation also runs in $O(\log n)$ time. The $evert(v)$ operation is particularly useful when dealing with unrooted trees, by allowing the choice of an arbitrary node as a root, and then re-rooting as necessary using $evert(v)$.

New queries/operations. From now on, we assume that our tree is a rooted binary tree, where each node is either of degree 1 (a *leaf node*), of degree 2 (a *regular node*), or of degree 3 (an *internal node*). We now introduce a set of queries and operations on such a binary tree that can be implemented in terms of dynamic tree operations, and will be needed later in our algorithms. Each node has at most degree 3, so we store with each node 3 unique branch ids called $branchId$ to identify the potentially 3 different branches incident to it (in a rooted tree, a node's parent for example might be assigned branch id 1, and the two potential children would be assigned branch ids 2 and 3). Consequently we assume the availability of a function $getBranchId(u, v)$ which returns the $branchId$ that corresponds to the child/parent node v of u. We call two nodes u and v consecutive in the tree, if u and v are connected with a tree edge i.e. $u = parent(v)$ or $v = parent(u)$.

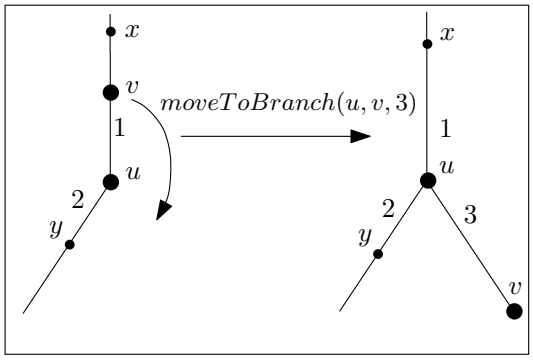


Figure 1: An example of moveToBranch. Before the call to $moveToBranch(u, v, 3)$, v belongs to branch id 1, and the call moves it to branch id 3 which did not initially exist, so it gets created.

- $swapNodes(u, v)$: Given **any two consecutive nodes** u **and** v swap the positions of u and v in the tree.
- $moveToBranch(u, v, branchId)$: Given **any two consecutive nodes** u **and** v, it will move node v from its current branch (relative to u) to u's branch $branchId$.

- $isInBranch(s, v)$: Given **an internal node** s **and any node** v, if v is contained in the subtree rooted at one of the children of s, then it returns the branch id of the corresponding subtree of s that contains v. Otherwise, it returns NULL.

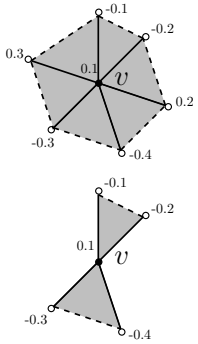
It is easy to check that each operation above can be implemented using constant number of dynamic tree operations. For example, the operation $isInBranch(s, v)$ is implemented as follows — since s has only two children as the input tree is a binary tree, $isInBranch(s, v)$ runs in $O(\log n)$ time.

Procedure $isInBranch(s, v)$
for all $c \leftarrow child(s)$ **do**
  $cut(c)$
  $r \leftarrow findroot(v)$
  $link(c, s)$
  if $r = c$ **then**
    return getBranchId(s, c)
  end if
end for
return null

3 Common Preliminaries

In this section we present the common terminology used for both persistent pairings and contour trees. In particular, both methods rely on Morse functions, and proper discrete setting, which we now describe. In Sections 4 and 5, further notations will be introduced that are specific to each individual problem at hand.

Morse functions on d-manifolds. Given a smooth d-manifold M and a smooth function $f : \mathrm{M} \to \mathbb{R}$. A point $p \in \mathrm{M}$ is called critical if the gradient of f at p vanishes, that is $\nabla f(p) = 0$. A critical point is non-degenerate if the Hessian matrix of second partial derivatives at this point is non-singular. A *Morse function* is a smooth function with no degenerate critical points, and where no two critical points share the same function value. Hence all critical points of a Morse function are isolated.



Discrete setting. In the discrete setting, a triangulation K of a d-manifold M is a simplicial d-complex K whose underlying space $|K|$ is homeomorphic to M. A scalar function f we consider in this discrete setting is a piecewise-linear (PL) function $f : |K| \to \mathbb{R}$ defined at the vertices of K and linearly interpolated within any other simplex of K. In the discrete setting, one can have a combinatorial definition of critical points compatible with that of the smooth case, first introduced in [2] and later refined in [15]. In order to do so, we first define the following terms: the *star* of a vertex v, denoted by $Str(v)$, is the set of all simplices containing v. The link of v, denoted by $Lnk(v)$, is the set of simplices from the closure of $Str(v)$ that do not contain v. See the top figure on the right for an example in the case of a 2-manifold, where the shaded regions and solid lines form the star of v, while the dashed lines and empty dots form the link of v. The *lower-star* of v, denoted by $LowStr(v)$, contains all the simplices in the star of v for which v has the highest function value, and the *lower-link* of v, denoted by $LowLnk(v)$, contains all the simplices in the link of v whose endpoints have lower function values than v. Similarly one can define the upper-star and upper-link of a vertex.

Given a PL-function f defined on a d-dimensional triangulation K, the star of every vertex $v \in K$ is homeomorphic to an open d-ball, while the link of v is topologically a d-sphere. We say that v is a *regular point* if both the lower-star and upper-star of v have trivial homology for any dimension $p \leq d$. Otherwise,

v is a critical point. Let $\beta_p(X)$ denote the p-th betti number of a topological space X, and $\tilde{\beta}_p(X)$ the p-th reduced betti number of X. In particular, $\tilde{\beta}_p(X) = \beta_p(X)$ for $p \geq 1$, and for $p = 0$ or -1, we have that $\tilde{\beta}_0(X) = \beta_0 - 1$ and $\tilde{\beta}_{-1} = 0$ unless the X is the empty set. An *index-p* critical point is a point v with $\tilde{\beta}_{p-1}(LowLnk(v)) \neq 0$. An index-$p$ critical point v is *non-degenerate* if $\beta_{p-1}(LowLnk(v)) = 1$, and $\beta_i(LowLnk(v)) = 0)$ for all $i \neq p$. See the figure in the previous paragraph for an example of a non-degenerated index-1 critical point (saddle point). Finally, a Morse function in the continuous case refers to a PL-function such that: (i) all critical points have different function values; and (ii) no critical point is degenerate.

In this paper we are interested in the triangulation of 2-manifolds for the persistence case and of 3-manifolds (in fact, of the 3-sphere $\mathbb{S}^3$) for the contour tree case. We will further discuss the particulars of critical points for each case in its respective section later.

4 Time-Varying Persistence Pairings

4.1 Background

Persistent homology was first introduced by Edelsbrunner et al. [17]. See the book [13] for more detailed discussions as well as recent development / extensions of the concept. At the core of the persistence algorithm lies the idea of tracking the changes of homology groups with respect to a *filtration*. Specifically, given a topological space represented as a simplicial complex K, a filtration is a nested and ordered sequence of sub-complexes: $\emptyset = K_0 \subset K_1 \subset \ldots \subset K_m = K$. Let $H_p(K_i)$ denote the p-th homology group of the complex K_i. During the filtration, a new homology class h can be created in some sub-complex K_i through the introduction of a certain simplex α to K_{i-1} (that is, $h \in H_p(K_i)$ does not have pre-image under the inclusion map from $H_p(K_{i-1})$ to $H_p(K_i)$). That homology class could be killed at a later point in K_j through the introduction of another simplex β to K_{j-1} (that is, under inclusion the image of h is non-trivial in $H_p(K_{j-1})$ but becomes zero in $H_p(K_j)$ for *some* p). The simplex responsible for the birth of a homology class is *paired* with the the simplex that kills it, i.e. α is paired with β, and the *persistence* of the corresponding homology class is defined as $j - i$. Thus persistence measures the topological lifespan of homology classes in the input filtration. See [17] for more formal definitions and descriptions.

In many situations, a filtration is induced by a scalar function f defined on a manifold M. Specifically, let M_a denote the sub-level set $M_a = \{p \in M \mid f(p) \leq a\}$, then a sequence of real numbers $a_1 < a_2 < \cdots < a_n$ induces a filtered space (filtration) $M_{a_1} \subseteq M_{a_2} \subseteq \cdots \subseteq M_{a_n} = M$, which we call a *sub-level set filtration*. Applying the persistence algorithm to this filtration can pair up critical points of f (instead of simplices)[1] and can shed lights on the behavior of f over M. For example, the Connolly function [8], used in the molecular biology community to measure cavities and protrusions over a molecular surface M, can be studied under persistence to measure the importance of the cavities and protrusions it reports [21].

The general persistence algorithm runs in $O(n^3)$ time [17] where n is the size of the input complex K. However, for the case where K is a triangulation of a 2-manifold and the filtration is induced by a piecewise-linear function on K, persistence pairing of critical points can be computed in $O(n\alpha(n))$ time (or $O(n \log n)$ time if vertices are not already sorted) by using the union-find data structure. We are interested in this paper to efficiently maintain the persistence pairing for a time-varying scalar function defined on a 2-manifold.

Persistence pairing for a 2-manifold and UF-trees. For a Morse function f defined on an orientable 2-manifold M, there are three types of critical points: minimum, saddle point, and maximum, of indices 0, 1, and 2, respectively. For the sub-level set filtration induced by f, other than the global minimum and

[1] More precisely, it will pair up critical values of f, which are values that a critical point of f can take. For simplicity, we assume that the pairing is between corresponding critical points for Morse functions.

global maximum which remain unpaired, the persistence algorithm will pair every minimum (resp. every maximum) with a saddle point. That is, a persistence pairing consists either of a minimum and a saddle or of a maximum and a saddle point. If $\beta_1(M) = 2g$ (i.e, the genus of M is g), then exactly $2g$ saddles remain unpaired, and we call them *essential saddles*. Computing persistence pairing for f means that we wish to identify all critical points of f, output all persistence pairings, as well as the essential saddles. Identifying critical points can be easily done by inspecting the lower-link of each vertex once. After we compute all persistence pairings, the remaining unpaired saddles are essential saddles. To compute persistence pairings for a 2-manifold, it turns out [17, 12] that one can sweep the manifold twice, once *upward* in increasing order of function values to compute all minimum-saddle pairings, and once *downward* in decreasing order of function values to compute all maximum-saddle pairings. These two sweeps are symmetric, so from now on, we only consider the upward sweep for minimum-saddle pairings.

Specifically, the minimum-saddle pairings are computed as follows: As one scans M upward, one maintains connected components in the sub-level set $M_a := \{p \in M \mid f(p) \leq a\}$ with the help of the union-find data structure. A minimum v corresponds to a $CreateSet(v)$ operation, as after scanning through v, a new component will be created in $M_{f(v)}$. If two connected components C_1 and C_2 merge at M_a, that means that we are passing through a saddle point s with $f(s) = a$. (Note, the inverse is not true. Passing through a saddle point may or may not corresponding to a merging event in the sub-level set.) Furthermore, the saddle point s will be paired with the higher global minima of C_1 and of C_2. The result of this process is a merge tree T which we refer to as a *UF-tree*, with the following property, : (see Figure **??** for an illustration.)

(1) All nodes in T are either of degree-1 and degree-3. Other than the root, all internal nodes are of degree-3. The root corresponds to the global maximum and also has degree-1.

(2) Each leaf node corresponds to a minimum of f, and every minimum of f corresponds to a leaf in the tree.

(3) For any isovalue a, the set of subtrees below the isovalue a represent the set of components in the sub-level set M_a.

(4) Take a subtree T_v rooted at node v. It corresponds to a connected component, say C_v, in the sub-level set $M_{f(v)}$. The lowest leaf node in T_v corresponds to the global minimum in C_v.

(5) Each degree-three node v corresponds to a saddle point s. The two child-subtrees rooted at v correspond to the two connected components C_1 and C_2 in M_a that are merged when scanning through s. Furthermore, the saddle point s is paired with the lower global minimum of C_1 and C_2.

In other word, once we have the UF-tree T, we can retrieve from it all minimum-saddle persistence pairings easily: a saddle s will be paired with the higher global minima of its two sub-trees. We will see that later, we maintain such a UF-tree through time in order to maintain persistence pairings for a time-varying function.

Discrete setting. We are now given a triangulation K of a 2-manifold. For simplicity, we assume that every edge uv in K satisfy the *link condition* [10], that is, the intersection $lnk(u) \cap lnk(v)$ contains exactly two vertices. If this condition does not hold for K, we can perform a level of barycentric subdivision of K to make the condition valid.

For a Morse function defined on a smooth 2-manifold M, there are three types of critical points: maximum, minimum, and saddle. In the discrete setting, for a PL function f defined on a triangulation K of M, a vertex of K is *regular* if both its lower-link and its upper-link have exactly one connected component; otherwise, it is a *critical point.* Specifically, a *maximum* is a vertex whose upper link is empty. A *minimum* has an empty lower-link. A vertex is a *saddle point* if it has two or more connected components in its lower-link, and it is *non-degenerate* if it has exactly two connected components in its lower-link.

4.2 Problem Definition and Algorithm Setup

In this paper, we consider the case where the input function $F : |K| \times \mathbb{R}^+ \to \mathbb{R}$ is a time-varying piecewise-linear function. That is, for any time $t \in \mathbb{R}^+$, $F_t := F(t, \cdot)$ is a piecewise-linear function on K. We assume that F is generic in the sense that F_t is a Morse function other than at finite discrete moments. For each specific F_t, the static algorithm described above can compute the set of persistence pairings for its critical points. As t changes, new critical points may emerge, old ones may disappear, and the persistence pairings may change. Our goal in this paper is to track changes in critical points and their persistence pairings.

In particular, we note that the persistence pairings rely only on the order of input vertices (not their function values) and their connectivity (which never changes). Hence for a time-varying function, persistence pairings can potentially change only at a *crossing event* when two input vertices switch the order of their function values. In [6], Edelsbrunner et al. showed that in the general case, one can update persistence pairing in $O(n)$ time per crossing event, where n is the size of the input simplicial complex. In this paper, we show that for a time-varying function defined on a 2-manifold, each crossing event can be processed in $O(\log n)$ time by maintaining the UF-tree using a set of operations on dynamic trees, Note that in this paper, we discuss only minimum-saddle persistence pairings using the upward UF-tree. The maximum-saddle pairings can be handled symmetrically using the downward UF-tree.

Augmenting a UF-tree. The nodes in the UF-tree as described in Section 4.1 contain all minima and only a subset of saddle points — these are the saddle points whose two lower-links are coming from two different connected components in the sub-level set. We call these saddles *merging saddles*. For a time-varying function, critical points change and regular points can become critical. Hence we want to keep track of all vertices from K. Specifically, we augment the UF-tree with a set of degree-2 nodes to represent all the remaining vertices of K, i.e., the non-merging saddles and the regular points, thus the resulting tree T, is a binary tree with nodes of degree 1, 2, and 3.

Maintaining lower links. During the updates, we routinely need to identify a vertex from each connected component of the lower-link of a given vertex v for the current function. Linear scanning to compute it takes time $O(|Lnk(v)|)$ for each vertex v, which can be linear in the worst case. Instead, for every vertex v, we maintain vertices in each component of its lower-link explicitly by a balanced tree with the id of each vertex being its key. Dynamically maintaining such trees take only $O(\log |Lnk(v)|)$ time under each insertion, deletion, merge, and split operation. With this tree, a vertex from the lower-link of a given vertex can be retrieved in $O(1)$ time.

Algorithm setup. We are given a triangulation K of a 2-manifold with n vertices, and a piecewise-linear time-varying function $F_t : |K| \times \mathbb{R}^+ \to \mathbb{R}$ on K. At the beginning of the algorithm, we perform the static persistence algorithm to compute the UF-tree T_0 corresponding to the sub-level set filtration induced by F_0. We assume that all crossing events are already given and stored in a priority queue sorted by the time they happen – these events can either be computed offline if F_t is known a prior; or they are generated online as time goes on. Our goal is to maintain the UF-tree after each crossing event. In order to retrieve the persistence pairings, we maintain that each saddle-node v stores the corresponding saddle point and its pairing minimum. Similarly, we store with each min-leaf the corresponding minimum and its pairing saddle.

For each vertex $p \in K$, we store with it a pointer to the corresponding UF-tree node. We also maintain each component in $LowLnk(p)$ using the data structure described before. From this lower-link information, we can retrieve the criticality of p in $O(1)$ time by checking how many components it has. From now on, for simplicity, we sometimes abuse the notation slightly and use p to refer both to a vertex in K and to its corresponding tree node.

4.3 Crossing Events

The key step we need to describe is how our algorithm processes a crossing event. In the rest of the paper, we use X^-, X, and X^+ to represent the value of a quantity X at time $t^- = t - \epsilon$, t, and $t^+ = t + \epsilon$, respectively, where ϵ is an infinitesimally small positive number. For simplicity, we also use f^-, f, and f^+ to denote F_{t^-}, F_t, and F_{t^+}, respectively. Now consider a crossing event between vertices $p, q \in K$ at time t (i.e, p and q switch their order at time t). Assume that $f^-(p) < f^-(q)$ but $f^+(p) > f^+(q)$ unless stated otherwise. We already have the UF-tree T^-, and we wish to update it to obtain T^+. Note that a crossing event may change the UF-tree structure itself, and it may also change the pairing information we associate with the nodes.

We distinguish two cases: (i) p and q are the endpoints of an edge e in the input mesh K; and (ii) p and q do not share an edge. We classify a crossing event by the criticality of p and q. Specifically, *'M'*, *'S'*, and *'R'* stand for *minimum*, *saddle*, and *regular* point, respectively.

4.3.1 Edge-connected crossing events

First, consider the case where p and q are endpoints of some edge from K. Based on the criticality of vertices p and q, there are potentially four types of events: eMS, eMR, eRR, and eSR [2]. Since we store the criticality information of each vertex, we can identify the type of event in $O(1)$ time. In all illustrations in this section, signs are relative to the vertex p.

Regular-regular (eRR) crossing: There are three cases as illustrated in Figure 2. In (a), p used to be the

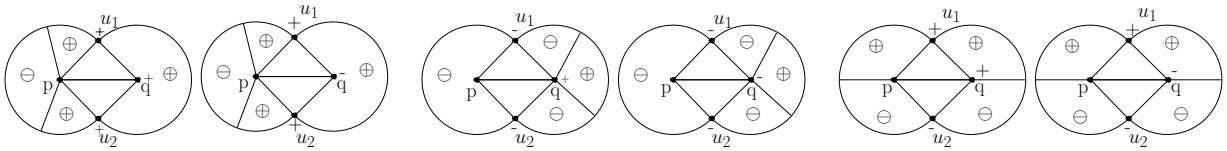
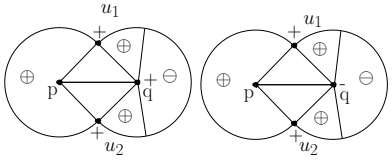


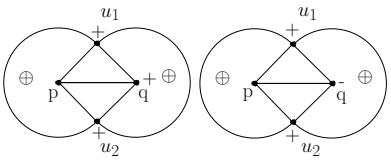
Figure 2: In (a), (b), and (c), the left and the right pictures show the configurations before and after the crossing event respectively. $\{u_1, u_2\} = lnk(p) \cap lnk(q)$, and the signs are relative to the vertex p.

only vertex in $LowLnk(q)$ $f^-(p) < f^-(q)$ at time t^-. At time t we have a *birth event*, where at t^+, a pair of critical points: a local minimum (q) and a saddle (p) are created. In the UF-tree, we have two consecutive regular nodes v_p, and v_q, turning into a pair of parent-child nodes, with v_p being the saddle-node and v_q be a min-leaf. This can be achieved by a call to $moveToBranch(p, q, branchId)$.The case in (b) is a symmetric birth event, where at time t^+, a local maximum p and a saddle q are created. In (c), nothing happens and p and q remain regular. In the UF-tree, the two consecutive regular simply swap places, achieved by a call to $swapNodes(p, q)$. To identify which of the three cases happen, we only need to inspect the link of edge pq, which is of constant complexity. Hence this event can be processed in $O(\log n)$ time.

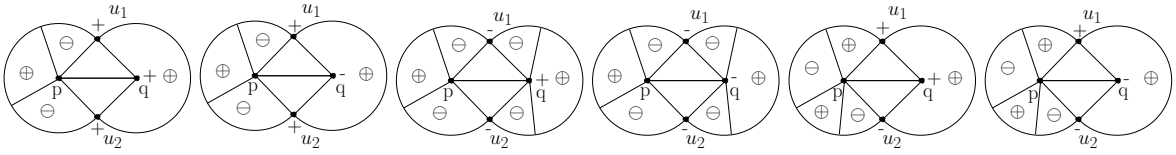


Minimum-saddle (eMS) crossing. In this case, a *death event* happens where p and q both become regular at time t^+. (This is the reverse of the birth-event in Figure 2 (a).) See the right figures which show the links of p and q at time t^- and t^+, respectively. At time t^+, q creates a component in $LowLnk(p)$ thus making p into a regular point. In the UF-tree, a parent-child pair of nodes become two consecutive regular nodes, which can be done by a call to $moveToBranch(p, q, branchId)$.Hence this event takes $O(\log n)$ time to process.

[2]The event eMM cannot happen. The event eSS can only happen for non-Morse functions hence we ignore this case.



Minimum-regular (eMR) crossing: Since we have assumed that $f^-(p) < f^-(q)$, p is the minimum at t^-. At t^+, $LowLnk(p) = \{q\}$ implying that p becomes a regular vertex. Symmetrically, the lower link of q becomes $LowLnk(q) = \emptyset$ at time t^+. Hence q becomes a minimum. In other words, the minimum shifts from p to q after this crossing event. See the right figure for an illustration. In the UF-tree, the two leaf nodes p and q exchange places, and we update T^- to T^+ by a call to $swapNodes(p, q)$, which takes $O(\log n)$ time. Suppose s was the pairing partner of p at time t^-. We also need to change the pairing partner stored at the tree node s to be q. This can be done in constant time.



(a) (b) (c)

Figure 3: (a) – (c): left and right pictures show configurations before and after crossing, respectively.

Saddle-regular (eSR) crossing. Assume without loss of generality that p is the saddle point and q is the regular point. There are three possibilities (see Figure 3 for illustrations). Let $\{u_1, u_2\} = lnk(p) \cap lnk(q)$.

1. Suppose both $f^-(u_1), f^-(u_2) > f^-(p)$ at time t^-. In this case, u_1, u_2 and q are in the same component of the upper link of p at time t^-. At time t^+, $f^+(p) > f^+(q)$ implying that $LowLnk(p)$ will obtain a new component $\{q\}$. This means that $LowLnk(p)$ has three components at t^+, making it a monkey saddle, and q becomes a maximum. This is a degenerate case that we do not consider.
2. Suppose both $f^-(u_1), f^-(u_2) < f^-(p)$ at t^-. Then at time t^+, $q \in LowLnk^+(p)$, connecting two components in $LowLnk^-(p)$ at time t^- into one component at time t^+. Hence p becomes a regular point, and q becomes a saddle. A call to $swapNodes(p, q)$ is sufficient which takes $O(\log n)$ time.
3. Suppose $f^-(u_1) > f^-(p)$ and $f^-(u_2) < f^-(p)$ at time t^-. In this case, p remains a saddle and q remains regular at time t^+, however q moves from the upper branch of p to one of the two lower branches of p. In order to determine which lower branch of p, q should move to, we perform a call to $isInBranch(p, u_2)$, and move q to the branch whose id is returned by $isInBranch$. To move q to $branchId$ we call $moveToBranch(p, q, branchId)$. Both operations take $O(\log n)$ time.

4.3.2 Vertices With No Common Edge

Based on the criticality of p and q, there are six potentially cases: nMR, nMS, nRR, nSR, nMM, and nSS. Since p and q are not connected by an edge, switching the order of p and q will not change the criticality of either vertex. It is easy to check that nMR and nMS will neither change the pairing of the minimum involved, nor will it change the position of any node in the tree. Hence there is nothing to update, and we only describe the remaining four cases below. From now on, in all illustrations, we plot the graph of the function f; that is, the height of a vertex represents its function value.

Regular-regular (nRR) crossing: A change can only occur if the two vertices are in the same component, in which case they must be consecutive nodes along the UF-tree. This can be decided by checking whether $q = parent(p)$ in $O(1)$ time. If p and q are consecutive, then we perform $swapNodes(p, q)$ to swap the positions of p and q in the UF-tree in $O(1)$ time.

Saddle-regular (nSR) crossing: This case is similar to the eSR case. The pairing will not change, but some node may change its position in the UF-tree. Assume without loss of generality that p is a saddle vertex and q is a regular vertex. If p is not a merging saddle, then the handling of this crossing event is the same as that of nRR. Now assume that p is a merging saddle, thus p has two lower-branches in the UF-tree.

We again distinguish the cases where $f(p) > f(q)$, and $f(p) < f(q)$. In both cases, a change in the UF-tree can only happen if p and q are in the same connected component before or after the event. This can be identified by checking whether $p = parent(q)$ if $f(p) > f(q)$ or checking whether $q = parent(p)$ if $f(p) < f(q)$. In the first case, q moves from a component (subtree) rooted at p to the branch (with branch id b_1 for example) above p , and a call to $moveToBranch(p, q, b_1)$ is sufficient to update the UF-tree.

In the second case, q will move from the upper branch of p to one of the two lower-branches of p, and we need to determine which branch q will move to. To do this, we take one vertex, say u, from q's lower link in $O(1)$ time, and determine which child-subtree of p the vertex u belongs to — obviously, q will be in the same component (and thus branch in the tree) as u at time t^+. This can be achieved by a call to $isInBranch(p, q)$ which will return the branch id $branchId$ of p whose subtree contains u. Moving q to branch $branchId$ is done by a call to $moveToBranch(p, q, branchId)$. The entire process takes $O(\log n)$ time.

Min-min (nMM) crossing: In this case, the topology of the UF-tree will not change. However, the minima involved may swap their pairing partners. In particular, consider the UF-tree T^- at time t^-. The minimum p and q will be in different components at the beginning, until we sweep through some saddle point s that merge the two components C_p and C_q containing p and q, respectively. Note that p (resp. q) may or may not be the global minimum for the component C_p (resp. C_q).

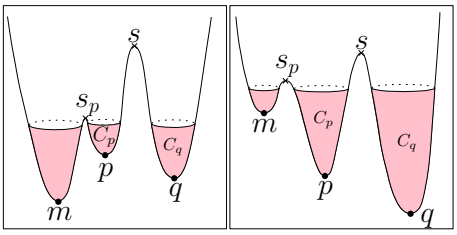


Figure 4: left: no change. right: change

If one of p and q, say p, is already paired with say s_p in C_p, then neither of them will change their pairing partner when p and q switch their order at time t^+. See Figure 4 (b) for an example. This is because for function f^+, at the time we sweep the saddle s_p, q is still in the different component from p and s_p in the sub-level set. So s_p will still be paired with p for function f^+. A change in pairing will only occur if both p and q compete for the same saddle s. In particular, neither p nor q was paired in C_p or C_q for the function f^-. This would imply that p and q will be the global minimum of C_p and C_q, respectively for both functions f^- and f^+. The saddle point s, by merging components C_p and C_q, will be paired with the higher global minimum of the two, say p at time t^-. Hence an order-switch of p and q at time t will change the pairing partner of s from p to q at time t^+. See Figure 4 for an example of no pairing change and an example of when there is a pairing change.

To algorithmically check whether a pairing change as described above happens or not, assume that $f^-(p) > f^-(q)$. Let s_p be the saddle paired with p and s_q the saddle paired with q at time t^-. If s_p is indeed the saddle s that we introduced above, that means (i) p and q must be in different components in the sub-level set below s_p; and (ii) $f(s_p) < f(s_q)$ to guarantee that q is the global minimum in its component when we sweep through s_p. Condition (i) can be checked by performing constant number of $isInBranch(s_p, q)$ to see which lower-branch of s_p that p and q are in. This takes $O(\log n)$ time. Checking condition (ii) takes only $O(1)$ time. Hence it takes $O(\log n)$ total time to process an nMM crossing event.

Saddle-Saddle (nSS) crossing:

Recall that only merging-saddles appear as degree-3 nodes in the UF-tree resulting from an upward sweep. Assume without loss of generality that $f^-(p) < f^-(q)$. First suppose p is not a merging saddle at time t^-. This means that the two components in the lower-link $LowLnk(p)$ of p are already connected in the sub-level set by the time we sweep through p. It is easy to see that this remains true at time t^+ no

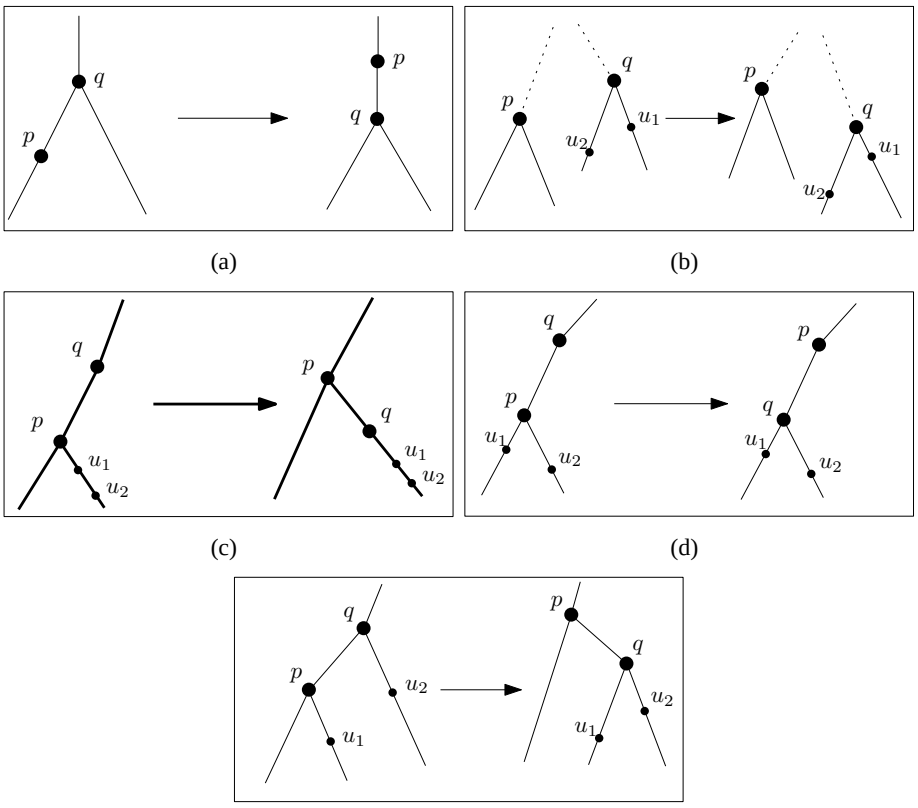


Figure 5: (a): p a non-merging saddle crossing q a merging saddle. (b): p and q, two saddles that belong to two different components in the sublevel set $M_{f(q)}$. (c) q is a non-merging saddle with both lower links in one of p's merged components. (d) q a non-merging saddle with lower links in both of p's merged components. (e) p and q are merging saddles.

matter what type of saddle q is, and in the UF-tree, we only need to move p to the upper-branch of q (whose branch id is $branchId$). This can be done by calling $moveToBranch(q, p, branchId)$ in $O(\log n)$ time. See Figure 5 (a).

The remaining case is that p is a merging saddle for the upward sweep. Assume p merges two connected components C_1 and C_2 at time t^-, whose corresponding subtrees are rooted at the children r_1 and r_2 (with branch ids b_1 and b_2) of the UF-tree node p, respectively. Now consider the saddle point q, and let u_1 and u_2 be two arbitrary points, one from each of the two components in the lower-link of q, respectively. We check whether u_1 and u_2 are contained in any child-subtree rooted at the saddle-node p. This is equivalent to testing whether u_1 and u_2 are from the components C_1 and C_2 merged by the saddle point p, and can be done by calling $isInBranch(p, \{u_1, u_2\})$ constant number of times in $O(\log n)$ time. There are four cases. See Figure 5 for an illustration.

(1) Neither u_1 nor u_2 is contained in the subtree rooted at p. This means that the saddle points p and q are not yet connected in the sub-level set when sweeping p or q at time t^-. Hence their persistence pairings are independent and will not be affected by switching the order of p and q. In the UF-tree, p and q are not consecutive and there is nothing to update either. See Figure 5 (b).

(2) Both u_1 and u_2 are contained in the same component, say C_1, at time t^-, implying that q is a non-merging saddle at time t^-. At time t^+, when we sweep through q, u_1 and u_2 are still contained in the same connected component C_1. Hence q remains a regular node, but it will move from the upper branch of p to the child-branch of p rooted at r_1 with branch id b_1. We simply need to perform a $moveToBranch(p, q, b_1)$ operation to update the UF-tree. See Figure 5 (c).

(3) Both u_1 and u_2 are contained in the subtree rooted at p, but each belongs to a different child-subtree. Specifically, assume that $u_1 \in C_1$ and $u_2 \in C_2$. In this case, the components C_1 and C_2 will be merged by either p or q, whichever comes first. In other words, the lower saddle among p and q will be a merging saddle and be paired with the higher global minimum of C_1 and C_2; while the higher one will be a non-merging saddle. Hence before and after this crossing event, the role of p and q will switch. The UF-tree can be updated in $O(1)$ time easily.

(4) Only one of them, say u_1, is contained in one of the component, say C_1; denote by C_q the component containing u_2 before sweeping p. This means that at time t^-, q will merge two connected components; $C_1 \cup C_2 \cup \{p\}$ and C_q. After p and q switch order, we will first sweep through q, which will merge the components C_1 with C_q. The saddle p will then merge $C_1 \cup C_q \cup \{q\}$ with C_2. Reorganizing the tree to reflect these changing in the merging pattern involves constant number of cut and link operations in $O(\log n)$ time. See Figure 5 (d). The persistence pairings of p and q may also change. In particular, let m_1, m_2, and m_q denote the global minima of components C_1, C_2, and C_q, respectively. Recall that a merging saddle will be paired with the higher minima from the two components it merges. Hence q will change its pairing partner if the global minimum of $C_1 \cup C_2 \cup \{p\}$ is different from that of C_1 (implying that m_1 is higher than m_2), and also the global minimum m_1 of C_1 is higher than m_q of C_q (otherwise, q will stay paired with the higher minimum m_q). Intuitively, p and q will compete for m_1 in this case, and m_1 will be paired with whoever merges component C_1 first. See Figure 5 (d) for an illustration. In summary, if m_1 is highest among m_1, m_2 and m_q, then the pairing partner of q changes from the higher of m_q and m_2 to m_1 after the crossing event, while that of p changes from m_1 to the higher of m_q and m_2. Computing m_1, m_2 and m_q, as well as updating persistence pairings if necessary, only takes constant time.

Putting everything together, the total time complexity for handling the nSS crossing event is $O(\log n)$. We summarize the main result of this section in the following theorem:

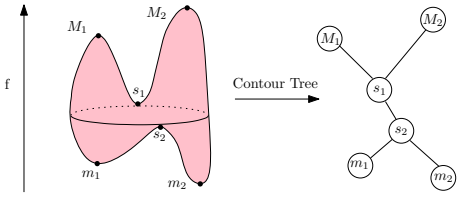
Theorem 1. *Given a triangulation K of a 2-manifold and a time-varying piecewise-linear function $F : |K| \times \mathbb{R}^+ \to \mathbb{R}$ defined on K, let n be the size of K. After spending $O(n \log n)$ time to build the persistence*

pairings for the function F_0, one can update the persistence pairings for F_t in $O(\log n)$ time, after each crossing event.

5 Contour Trees

5.1 Background

Contour tree. Suppose we are given a continuous function $f : X \to \mathbb{R}$ defined on a domain X. As we vary a value $\alpha \in \mathbb{R}$, the connected components in the level set $f^{-1}(\alpha)$ may appear, disappear, split and merge. The contour tree is an abstract tree structure that can track such changes. In particular, call each connected component of a level set *a contour*. Two points $x, y \in X$ are *equivalent* if they belong to the same contour. The contour tree of f, denoted by $\mathcal{T}_f$, is the quotient space induced by this equivalence relation.



Intuitively, $\mathcal{T}(f)$ is obtained by continuously collapsing each contour in the level set into a single point. One can imagine that there is a surjection map $\phi : X \to \mathcal{T}_f$ such that $\phi(x) = \phi(y)$ if and only if x and y are from the same contour. Note that ϕ induces a continuous function $\tilde{f} : \mathcal{T}_f \to \mathbb{R}$ on the contour tree, where $\tilde{f}(z)$ is defined as the function value of any point from the pre-image of z. Since $\phi^{-1}(z)$ is a contour and thus all points in it have the same f value, $\tilde{f}$ is well-defined. From now on, we abuse the notation slightly and use f to also denote $\tilde{f}$. See the right figure for an example — Note that the contour tree is an abstract graph. However in this paper, we always plot the contour tree so that the height of a point z corresponds to $f(z)$. Hence we can talk about that the *up-degree, down-degree, up-branch(es), and down-branch(es)* of a node $v \in \mathcal{T}_f$.

Contour tree for Morse functions. For a Morse function $f : M \to \mathbb{R}$ defined on a d-manifold M, each degree-one node v in $\mathcal{T}_f$ corresponds to either a minimum (when v has only one up-branch) or a maximum (when v has only one down-branch). The remaining nodes have degree-3, and there are two types: the one with down-degree 2 corresponds to a *downfork-saddle* that merges two contours in the level set $f^{-1}(\alpha)$ as we increase the function value α; while the one with up-dedgree 2 corresponds to a *upfork-saddle* that split a single contour into two components in $f^{-1}(\alpha)$ as we increase α. For a d-manifold, a downfork-saddle is an index-1 saddle point, or 1-*saddle* for short; while a upfork-saddle is an index-$(d-1)$ saddle point (i.e, a $d-1$-*saddle*).

In this paper, we will consider a Morse function f defined on a 3-manifold. Hence there are four types of critical points of indices 0 to 3: miminum, 1-saddle, 2-saddle, and maximum. A subset of 1-saddles (resp. 2-saddles) correspond to downfork-saddle nodes (resp. upfork-saddle nodes) in the contour tree $\mathcal{T}_f$. Since M is simply connected, it turns out that a 1-saddle s is a downfork-saddle if and only if the two components in the lower-link of s are not connected in the **sub-level set** $\{x \in M \mid f(x) < f(s)\}$ [3]. Similarly, a 2-saddle s is a upfork-saddle if and only if the two components in the upper-link of s are not connected in the **sup-level set** $\{x \in M \mid f(x) > f(s)\}$.

5.2 Algorithm setup

In this paper, we consider a simplicial 3-complex K whose underlying space is homeomorphic to a 3-sphere, and an input piecewise-linear, time-varying function $F : |K| \times \mathbb{R}^+ \to \mathbb{R}$. We assume F is generic in the sense that $F_t := F(t, \cdot)$ is a discrete Morse function other than discrete choices of t (when

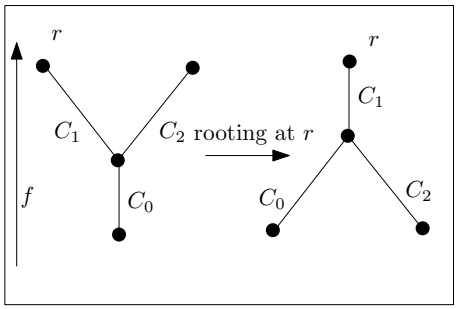
[3] By definition, a downfork-saddle merges two components in the **level set**. For simply connected domain, it turns out that this is equivalent to merging two components in the **sub-level set**.

a crossing event happens). We first compute the contour tree for F_0 using the algorithm by Carr et al. [5] in $O(n \log n + N\alpha(N))$ time, where n is the number of vertices and N is the number of simplices in the 2-skeleton of K (since the contour tree depends only on the 2-skeleton of the input simplicial complex). Our goal is to maintain the contour tree $\mathcal{T}_t$ of F_t as the function changes. Similar to the case of persistence pairings, the contour tree structure is completely decided by the order of vertices in K with respect to the input function values. Hence we only need to potentially update the contour tree when there is a crossing event.

An augmented contour tree. Similar to Section 4.2, we augment the contour tree by inserting a degree-2 node for all the remaining vertices from K (i.e, vertices other than minima, maxima, upfork-saddles and downfork-saddles). We define *consecutive nodes* along this binary tree as before in Section 4.2.

We note that the contour tree is similar to the UF-tree we used before. Indeed, the contour tree is the combination of both the upward and the downward UF-tree [4]. For the upward UF-tree T in the previous section, we use the global maximum as the root of T, and this natural choice guarantees that the children of a saddle point always correspond to the two components it merges; while its parent node always correspond to the merged component. For the contour tree, we can adopt the same choice of using the global maximum as the root.

However, the children of a upfork-saddle s no longer correspond to the two components that s splits into. See the right figure for an example. From now on, for simplicity, we still draw all the tree such that the height of a point represent its f function value, and use "lower branch(es)" and "upper branch(es)" to refer to their order with respect to this height function.



Finally, we observe that as the input function changes continuously, the global maximum may sometimes jump to a node in the contour tree not consecutive to the old global maximum. When that happens, we perform the $evert$ dynamic tree operation in $O(\log n)$ time to re-root the tree to the new maximum. The global maximum can be maintained by a standard priority queue in $O(\log n)$ time per crossing event. Hence from now on, we will skip discussing the update of global maximum.

Maintaining lower-link and upper-link vertices. Given a vertex $p \in K$, we now abuse the notation slightly and let $LowLnk(p)$ (resp. $UpperLnk(p)$) denote the 1-skelenton of the upper (resp. lower) link of p. Hence $LowLnk(p)$ (resp. $UpperLnk(p)$) is a graph. For each vertex $p \in K$, we need to maintain the connectivity of both the upper and lower links of p. We do this by using one dynamic graph data structure to maintain the spanning forest in $LowLnk(p)$, and one for $UpperLnk(p)$, respectively. The intersection of two links of vertices of a 3-sphere is a 2-sphere which is a compactification of a planar graph. To maintaining the forest of a planar graph G can be done in time $O(\log k)$ where k is the number of edges in the graph [18]. With this information, we can determine the criticality of the vertex p in constant time (by inspecting the number of components in the lower and upper link of p). We can also retrieve a vertex from each component in the lower or upper link of p in constant time.

We now look at the time complexity to update this data structure when a crossing event, say between p and q, happens. Observe that this crossing event can potentially change the lower-link and upper-link of vertices p and q only when p and q are connected by an edge in K. Assume that $f^-(p) < f^-(q)$. When p and q switches order, q will move from the upper-link of p to its lower-link. Removing q from $UpperLnk(p)$ means that we need to remove all edges from $UpperLnk(p)$ that are incident to q, and the number of such edges is at most $|Lnk(q) \cap UpperLnk(p)|$. Inserting q to $LowLnk(p)$ means that we insert all edges from $LowLnk(p)$ that are incident to q, which is bounded by $|Lnk(q) \cap LowLnk(p)|$. Hence the

[4]The algorithm by Carr et al. computes the contour tree by merging the two UF-trees [5].

total time complexity to maintain the spanning forest for lower/upper links of p and q takes $O(|Lnk(q) \cap Lnk(p)| \log n)$ if pq is an edge in K. If the crossing event happens between non-edge connected vertices p and q, then there is no update in the lower-link and upper-link data structures.

5.3 One crossing event

We now consider updating the contour tree for one crossing event between vertices p and q at time t. We use the same notations t^-, t^+, f^- and f^+ as in Section 4.3. Assume that we already have the contour tree $\mathcal{T}^-$, and we wish to update it to $\mathcal{T}^+$ for function f^+. First, note that a crossing event will have no effect on the contour tree unless p and q are consecutive nodes along the contour tree — as otherwise, p and q are in different contours at time t and their relative order does not affect what they do to the contours in the level set. Hence from now on, we assume that p and q are consecutive along the contour tree.

We distinguish the crossing events based on the criticality of p and q. There are four types of critical points: Let 'Min', 'Sad1', 'Sad2', and 'Max' represent minimum, 1-saddle, 2-saddle, and maximum, respectively. Let 'Reg' denote regular point. This gives us fifteen combinations. However, cases involving Max are symmetric to those involving Min. Similarly, cases involving Sad2 are symmetric to those involving Sad1. Furthermore, Min-Min, Min-Sad2 and Min-Max crossing events cannot happen — This is because if p is a minimum, then p and q are consecutive along the contour tree means that one component in the lower-link of q consists only of p. Hence q cannot be a minimum, an index-2 saddle point, nor a maximum, as in those cases $LowLnk(q)$ would consist of a empty set, a topological circle or a topological sphere, respectively. Therefore we only need to consider the following cases: Reg-Reg, Min-Reg, Min-Sad1, Sad1-Reg, Sad1-Sad1, and Sad1-Sad2 crossing events. Determining the type of a crossing event takes only constant time by checking the criticality of p and q from the spanning forests that we maintain for the lower-link and upper-link of a vertex. Given the similarity between the UF-tree and the contour tree, handling these events are quite similar to the processing of crossing events in Section 4.3.

Algorithm for time-varying contour tree by Mascarenhas et al. [14]. Mascarenhas et al. proposed in [14] an algorithm to maintain the Reeb graph for a time-varying function defined on a triangulation on $\mathbb{R}^3$ (or the compactification $\mathbb{S}^3$ of $\mathbb{R}^3$). Since both $\mathbb{R}^3$ and the 3-sphere $\mathbb{S}^3$ are simply connected, the Reeb graph they consider is loop-free, and thus simply the contour tree. Their algorithm tracks and processes only *critical crossing events*, which are crossing events such that the two points involved are both critical either before or after the order-switching. As such, they have already enumerated the structural changes in the contour tree induced by these critical crossing events. In the description of our algorithm below, we will not repeat such enumeration, but simply giving references to the corresponding cases from [14], as well as providing a simple illustration for some cases. The main difference between our algorithm and the algorithm of [14] is the following: Since the algorithm from [14] does not track regular point, a key operation it uses is the so-called $\mathsf{PATH}(u)$ operation, which returns a monotone path connecting leaves of the contour tree that contains u. For example, PATH operation is used to identify the arc of the non-augmented contour tree that contains a regular point, which is needed in processing a birth event. It is also used to check whether a node belongs to a specific subtree or not, which is needed to for example decide which child-subtree of p the node q should move down to from the upper-branch of p By augmenting the contour tree with all regular points, our algorithm can address what the PATH operation is used for in $O(\log n)$ time. However, this is at the cost that we now need to process all crossing events, instead of only those critical crossing events.

5.3.1 Reg-Reg **crossing event**

If p and q are not connected by an edge, then their criticality will not change. We simply swap their positions along the contour tree. If they are connected by an edge, then p and q may become critical after the crossing which gives us a birth event. By the Index-Lemma from [14] a birth event can only produce a pair of critical

points with index differing by 1. Hence a pair of Min-Sad1, or Sad1-Sad2, or Max-Sad2 may be created. The handling of Min-Sad1 is the same as handling the eRR crossing event for the persistence pairing in Section 4.3. The handling of Max-Sad2 is symmetric to Min-Sad1. For Sad1-Sad2 birth event, as shown by Figure 5 from [14], a short-lived handle is created at the crossing. Afterwards, the newly created 1-saddle point p (resp. 2-saddle q) is a non-merging saddle (resp. non-splitting saddle); that is, it cannot be a downfork-saddle (resp. upfork-saddle). Hence we only need to swap the positions of p and q along the contour tree.

5.3.2 Min-Reg **and** Min-Sad1 **crossing events**

Assume that p is a minimum. Since p and q are consecutive in the contour tree, p must be the only node in one of the lower-branch connected to q. This implies that p and q must be connected by an edge from K. Hence the handling of Min-Reg and Min-Sad1 events are the same as the handling of the eMR and eMS crossing events for persistence pairings in Section 4.3, respectively. In particular, in the case that q is a regular point, afterwards, q will be the new minimum, while p will become a regular point. In the case that q is a 1-saddle point, a death event happens at time t and afterwards, both p and q will be regular.

5.3.3 Sad1-Reg **and** Sad1-Sad1 **crossing events**

As we mentioned earlier, for a simply connected domain, a 1-saddle merges two contours in the *level set* if and only if it merges two connected components in the *sub-level set*. In other words, a downfork-saddle in the contour tree is equivalent to a merging-saddle in the upsweep UF-tree. Hence the changes in the contour tree induced by Sad1-Reg (resp. Sad1-Sad1) crossing event are the same as the changes in the UF-tree for eSR and nSR (resp. eSS and nSS) events in Section 4.3. The only difference is that we no longer need to test for changes in the persistence pairings as discussed in Case (4) when handling nSS crossing event. The different cases involved in handling Sad1-Sad1 are also equivalent to cases (1a), (2a), (2b) and (4) enumerated in [14].

5.3.4 Sad1-Sad2 **crossing event**

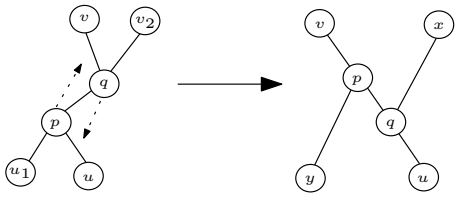
This is the main new case different from the updates of UF-tree as described in Section 4.3. Assume without loss of generality that p is a 1-saddle and q is a 2-saddle. We distinguish four cases.

(i) p is not a downfork-saddle, and q is not an upfork-saddle. That is, p is not merging, nor is q splitting. A death-event may happen when p and q are connected by an edge from K. (See Figure 5 from [14].) Otherwise, p and q maintain their criticality (case (1b) from Figure 6 in [14]). In either case, in the contour tree, p and q simply swap their positions.

(ii) p is a downfork-saddle, but q is not ans upfork-saddle. After the crossing, p remains a downfork-saddle and q is still not splitting. The only change is that q moves from the upper-branch incident to p to one of its lower-branches (or vice versa). To decide which lower-branch of p we should move q to, we choose a vertex v from the lower link of q and perform two $isInBranch(p, v)$ operations to identify the lower-branch that v is from. We then move q to that branch using $moveToBranch(p, q, branchId)$. The process takes $O(\log n)$ time. This scenario corresponds to case (2c) from Figure 6 in [14].

(iii) p is not a downfork-saddle, but q is a upfork-saddle. This is symmetric to the above case, and it corresponds to case (3c) from Figure 6 [14].

(iv) p is a downfork-saddle and q is a upfork-saddle. This corresponds to case (5) from Figure 6 in [14]. After the crossing, p and q remain a downfork-saddle and an upfork-saddle, respectively, but we need to re-arrange the contour tree locally. In particular, we choose a vertex u from the lower-link of q, as well as a vertex v from the upper-link of p, and perform constant number of $isInBranch$ operations to decide which branch of p (resp. q) we should move q (resp. p) to. Rearranging the contour tree takes only constant number of cut and $link$ operations. Hence this case can be processed in $O(\log n)$ time as well.



example of down-fork-up-fork crossing

Putting everything together, we have the following result for this section.

Theorem 2. *Given a triangulation K of $\mathbb{S}^3$ and a time-varying piecewise-linear function $F : |K| \times \mathbb{R}^+ \to \mathbb{R}$ defined on K, let n and N denote the number of vertices, and the size of 2-skelenton of K, respectively. After spending $O(n \log n + N\alpha(N))$ time to build the initial contour tree for function F_0, one can process each crossing event between vertices $p, q \in K$*

(i) in $O(|Lnk(p) \cap Lnk(q)| \log n)$ time if p and q are connected by an edge;

(ii) in $O(\log n)$ time otherwise.

References

[1] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *Proceedings of the 1996 symposium on Volume visualization*, VVS '96, pages 39–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[2] T. F. Banchoff. Critical points and curvature for embedded polyhedral surfaces. *American Mathematical Monthly*, pages 475–485, 1970.

[3] K. G. Bemis, D. Silver, P. A. Rona, and C. Feng. A methodology for plume visualization with application to real-time acquisition and navigation. *Visualization Conference, IEEE*, 0:35, 2000.

[4] G. Carlsson, T. Ishkhanov, V. Silva, and A. Zomorodian. On the local behavior of spaces of natural images. *Int. J. Comput. Vision*, 76(1):1–12, 2008.

[5] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 918–926, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[6] D. Cohen-Steiner, H. Edelsbrunner, and D. Morozov. Vines and vineyards by updating persistence in linear time. In *Proc. 22nd Annu. ACM Sympos. Comput. Geom.*, pages 119–126, 2006.

[7] A. Collins, A. Zomorodian, G. Carlsson, and L. J. Guibas. A barcode shape descriptor for curve point cloud data. *Computers & Graphics*, 28(6):881 – 894, 2004.

[8] M. L. Connolly. Measurement of protein surface shape by solid angles. *J. Mol. Graphics*, 4:3 – 6, 1986.

[9] V. de Silva and G. Carlsson. Topological estimation using witness complexes. In M. Alexa and S. Rusinkiewicz, editors, *Eurographics Symposium on Point-Based Graphics*. The Eurographics Association, 2004.

[10] T. K. Dey, H. Edelsbrunner, S. Guha, and D. V. Nekhayev. Topology preserving edge contraction. *Publ. Inst. Math. (Beograd) (N.S*, 66:23–45, 1998.

[11] H. Edelsbrunner and J. Harer. Jacobi sets of multiple morse functions. *Foundations of Computational Mathematics*, 2004.

[12] H. Edelsbrunner and J. Harer. Persistent homology — a survey. *In Twenty Years After, eds. J. E. Goodman, J. Pach and R. Pollack, AMS.*, 2007.

[13] H. Edelsbrunner and J. Harer. *Computational topology: an introduction*. American Mathematical Society, 2010.

[14] H. Edelsbrunner, J. Harer, A. Mascarenhas, and V. Pascucci. Time-varying reeb graphs for continuous space-time data. In *Symposium on Computational Geometry*, pages 366–372, 2004.

[15] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Morse-smale complexes for piecewise linear 3-manifolds. In *Proceedings of the nineteenth annual symposium on Computational geometry*, SCG '03, pages 361–370, New York, NY, USA, 2003. ACM.

[16] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. 41th Annu. IEEE Sympos. Found. Comput. Sci.*, page 454, 2000.

[17] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28(4):511–533, 2002.

[18] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. R. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *J. Algorithms*, 13(1):33–54, March 1992. Special issue for 1st SODA.

[19] A. Gyulassy, V. Natarajan, V. Pascucci, P.-t. Bremer, and B. Hamann. Topology-based simplification for feature extraction from 3d scalar fields. In *Proc. IEEE Conf. Visualization*, pages 535–542, 2005.

[20] N. Max, R. Crawfis, and D. Williams. Visualization for climate modeling. *IEEE Comput. Graph. Appl.*, 13(4):34–40, 1993.

[21] V. Natarajan, Y. Wang, P.-T. Bremer, V. Pascucci, and B. Hamann. Segmenting molecular surfaces. *Comput. Aided Geom. Des.*, 23:495–509, August 2006.

[22] V. Pascucci and K. Cole-McLaughlin. Efficient computation of the topology of level sets. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 187–194, Washington, DC, USA, 2002. IEEE Computer Society.

[23] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, New York, NY, USA, 1981. ACM.

[24] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[25] N. Thune and B. Olstad. Visualizing 4-d medical ultrasound data. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 210–215, 1991.

[26] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 212–220, New York, NY, USA, 1997. ACM.

[27] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13:330–341, March 2007.