Propagate and Pair: A Single-Pass Approach to Critical Point Pairing in Reeb Graphs

Junyi Tu[1][0000−0001−7026−7454], Mustafa Hajij[2], and Paul Rosen[1][0000−0002−0873−9518]

[1] University of South Florida, Tampa FL 33620, USA
{junyi,prosen}@mail.usf.edu
[2] The Ohio State University, Columbus 43210, USA

Abstract. With the popularization of Topological Data Analysis, the Reeb graph has found new applications as a summarization technique in the analysis and visualization of large and complex data, whose usefulness extends beyond just the graph itself. Pairing critical points enables forming topological fingerprints, known as persistence diagrams, that provides insights into the structure and noise in data. Although the body of work addressing the efficient calculation of Reeb graphs is large, the literature on pairing is limited. In this paper, we discuss two algorithmic approaches for pairing critical points in Reeb graphs, first a multipass approach, followed by a new single-pass algorithm, called Propagate and Pair.

Keywords: Topological Data Analysis · Reeb graph · critical point pairing

1 Introduction

The last two decades have witnessed great advances in methods using topology to analyze data, in a process called Topological Data Analysis (TDA). Their popularity is due in large part to their robustness and applicability to a variety of domains [17]. The *Reeb graph* [21], which encodes the evolution of the connectivity of the level sets induced by a scalar function defined on a data domain, was originally proposed as a data structure to encode the geometric skeleton of 3D objects, but recently it has been re-purposed as a tool in TDA. Beside their usefulness in handling large data [12], Reeb graphs and their non-looping relative, contour trees [5], have been successfully used in feature detection [24], data reduction and simplification [7, 22], image processing [16], shape understanding [2], visualization of isosurfaces [3] and many other applications.

One challenge with Reeb graphs is that the graph may be too large or complex to directly visualize, therefore requiring further abstraction. Persistent homology [13], parameterizes topological structures by their life-time, providing a topological description called the *persistence diagram*. The notion of persistence can be applied to any act of birth that is paired with an act of death. Since the Reeb graph encodes the birth and the death of the connected components of the level sets of a scalar function, the notion of persistence can be applied to critical points in the Reeb graph [1]. *The advantage of this approach is simplicity and scalability—a large Reeb graph can be reduced to a much easier to interpret scatterplot.* Fig. 1 shows an example, where a mesh with a scalar function (Fig. 1(a)) is converted into a Reeb graph (Fig. 1(b)). After that, the critical points are
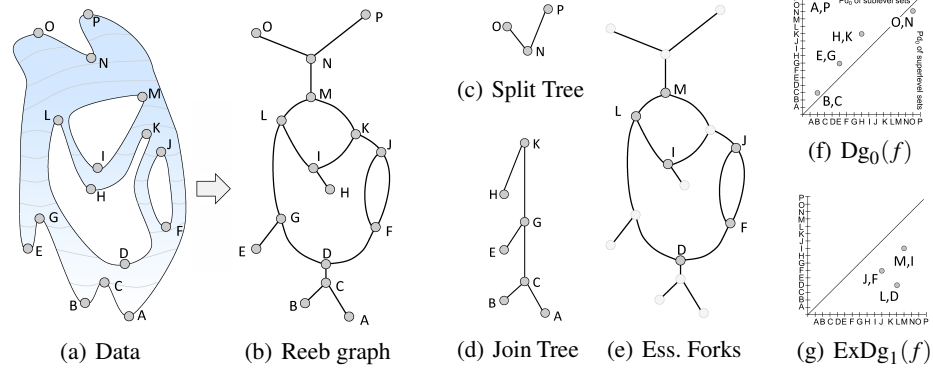
| (a) Data | (b) Reeb graph | (c) Split Tree (d) Join Tree (e) Ess. Forks | (f) $\mathrm{Dg}_0(f)$ (g) $\mathrm{ExDg}_1(f)$ |

Fig. 1. (a) A mesh with a scalar function being processed into (b) a Reeb graph, where critical points are paired. In the multipass approach, (c) a split tree and (d) a join tree are first extracted for non-essential pairing. Next, the (e) essential forks are paired, one at a time. (f) The persistence diagram and (g) extended persistence diagram provide a visualization of the pairings.

paired, and the persistence diagram displays the data (Fig. 1(f) and 1(g)). This final step can be challenging, particularly when considering *essential critical points*—those critical points associated with cycles in the Reeb graph—that each require an expensive search. While many prior works [8–11, 14, 15, 18, 20, 25] have provided efficient algorithms for the calculation of Reeb graph structures, to our knowledge, none have provided a detailed description of an algorithm for pairing critical points.

In this paper, we describe and test 2 algorithms to compute persistence diagrams from Reeb graphs. The first is a multipass approach that pairs non-essential (non-loop) critical points using branch decomposition [19] on join and split trees, then pairing essential critical points also using join trees. This leads to our second approach, an algorithm for pairing both non-essential and essential critical points in a single-pass.

2   Reeb Graphs and Persistence Diagrams

2.1   Reeb graph

Let X be a triangulable topological space, and let $f : X \to \mathbb{R}$ be a continuous function defined on it. The Reeb graph, R_f, can be thought of as a topological summary of the space X using the information encoded by the scalar function f. More precisely, the Reeb graph encodes the changes that occur to connected components of the level sets of $f^{-1}(r)$ as r goes from negative infinity to positive infinity. Fig. 1(a) and 1(b) show an example of a Reeb graph defined on a surface. For the sake of simplicity we plot the Reeb graph using the height function indicated by the vertical coordinate in the Figure.

The function f can be used to classify points on the Reeb graph as follows. Let x be a point in R_f. The *up-degree* of x is the number of branches (1-cells) incident to x that have higher values of f than x. The down-degree of x is defined similarly. A point x on R_f is said to be *regular* if its up-degree and down-degree are equal to one. Otherwise it

is a *critical point*. A critical point on the Reeb graph is also a *node* of the Reeb graph. A critical point is called a minimum if its down-degree is equal to 0. Symmetrically, a critical point is said a maximum if its up-degree is equal to 0. Finally, a critical point is said to be a down-fork/up-fork if its down-degree/up-degree is larger than 1.

2.2 Persistent Homology

The notion of persistent homology was originally introduced by Edelsbrunner et al. [13]. Here we present the theoretical setting for the computation of the persistence diagram associated with a scalar function defined on a triangulated topological space. Consider the p-dimensional homology class H_p of a space, where H_0 are components, H_1 are tunnels/cycles, H_2 are voids, etc. Persistent homology evaluates a sequence of vector spaces: $0 = H_p(X_0) \to H_p(X_1) \to \cdots \to H_p(X_n) = H_p(X)$, where $X_i = X_{\le f_i}$, recording the birth and death events. In particular, the p-th *ordinary persistence diagram* of f, denoted as $\mathrm{Dg}_p(f)$, is a multiset of pairs (b,d) corresponding to the birth b and death d values of some p-dimensional homology class.

Since the homology $H_p(X)$ may not be trivial in general, any nontrivial homology class of $H_p(X)$, referred to as an *essential homology class*, will never die during the sequence. These events are associated with the cyclic portions of the Reeb graph. We refer to the multiset of points encoding the birth and death time of pth homology classes created in the ordinary part and destroyed in the relative part of the sequence as the *p-th extended persistence diagram* of f, denoted by $\mathrm{ExDg}_p(f)$. In particular, for each point (b,d) in $\mathrm{ExDg}_p(f)$ there is an essential homology class in $H_p(X)$ that is born in $H_p(X_{\le b})$ and dies at $H_p(X_{\ge d})$. Observe that for the extended persistence diagram the birth time b for an essential homology class in $H_p(X_{\le b})$ is larger than or equal to death time d for the relative homology class in $H_p(X_{\ge d})$ that kills it.

2.3 Persistence Diagram of Reeb Graph

Of interest to us are the persistence diagram $\mathrm{Dg}_0(f)$ and extended persistence diagram $\mathrm{ExDg}_1(f)$. Pairing critical points can be computed independently of the Reeb graph. However, it is more efficiently computed by considering the Reeb graph R_f. We give an intuitive explanation here and refer the reader to [4] for more details.

First, we distinguish between 2 types of forks in the Reeb graph, namely the ordinary (non-essential) forks and the essential forks. Let R_f be a Reeb graph and let s be a down-fork such that $a = f(s)$. We say that the down-fork s is an *ordinary fork* if the lower branches of s are contained in disjoint connected components C_1 and C_2 of $(R_f)_{<a}$. The down-fork a is said to be *essential* if it is not ordinary. The ordinary and essential up-forks are defined similarly.

Ordinary Down-Forks of a Reeb Graph. We first consider pairing down-forks using sublevel set filtration. We track changes that occur in $H_0((R_f)_{\le a})$ as a increases. A connected component of $(R_f)_{\le a}$ is created when a passes through a minimum of R_f. Let C be a connected component of $(R_f)_{\le a}$. We say that a local minimum a of R_f *creates* C if a is the global minimum of C. Every ordinary down-fork is paired with a local minimum to form one point in the persistence diagram $\mathrm{Dg}_0(f)$ as follows. Let s be an

ordinary down-fork with $f(a) = s$ and let C_1 and C_2 be the connected components of $(R_f)_{<a}$. Let x_1 and x_2 be the creators of C_1 and C_2. Without loss of generality we assume that $f(x_1) < f(x_2)$. The homology class $[x_2]$ that is created at $f(x_2)$ and dies at $f(s)$ gives rise to a point (x_2, s) in the ordinary persistence diagram $\text{Dg}_0(f)$. Note, a pair occurs when the minimum is a branch in the Reeb graph, hence we name it a *branching feature*.

Ordinary Up-Forks of a Reeb Graph. Ordinary up-forks are paired similarly using superlevel set filtration, pairing each up-fork with a local maximum to form points in the persistence diagram, $\text{Dg}_0(f)$, with the following variations. For an ordinary up-fork, s, with $f(a) = s$, connected components C_1 and C_2 now come from $(R_f)_{>a}$. Assuming that $f(x_1) < f(x_2)$, the homology class $[x_1]$ that is created at $f(x_1)$ dies at $f(s)$ and gives rise to a point (x_1, s) in $\text{Dg}_0(f)$.

Cycle Features of a Reeb Graph. Let s be an essential down-fork. We call the down-fork s a creator of a 1-cycle in the sublevel set $(R_f)_{\leq a}$. As shown in [1], s will be paired with an essential up-fork s' to form an *essential pair* (s', s), and a point (s', s) in the extended persistence diagram $\text{ExDg}_1(f)$. The essential up-fork s' is determined as follows. Let Γ_s be the set of all cycles born at s, each corresponding to a cycle in R_f. Let γ_s be an element of Γ_s with largest minimum value of f among these cycles born at s. The point s' is the point that the function f achieves this minimum on the cycle γ_s.

3   Conditioning the Graph

Our approach is restricted to Reeb graphs where all point are either a minimum, maximum, up-fork with up-degree 2, or down-fork with down-degree 2. Fortunately, graphs that do not abide by these requirements can be conditioned to fit them. We define the $J : K$ degree of a node as the J up-degree and K down-degree.

There are 4 node conditions to be corrected: **1:1 nodes**—Nodes with both 1 up- and 1 down-degree are regular. Therefore, they only need to be removed from the graph. This is done by removing the regular point and reconnecting the nodes above and below, as seen in Fig. 2(a). **0:2 (and 2:0) nodes**—Nodes with 0 up-degree and 2 down-degree (or vice versa) are degenerate maximum (minimum) nodes, in that they are both down-fork (up-fork) and local maximum (minimum). As shown in Fig. 2(b), this condition is corrected by added a new node for the local maximum ε higher value, where ε is a small number. This type of degenerate node rarely occurs in Reeb graphs, but it frequently occurs in approximations of a Reeb graph, such as Mapper [23]. **2:2 nodes**—Nodes with both 2 up- and 2 down-degree are degenerate double forks, both down-fork and up-fork. Fig. 2(c) shows how double forks can be corrected by splitting into 2 separate forks, one up- and one down-fork, ε distance apart. **1:N>2 (and N>2:1) nodes**—Nodes with down-degree (or up-degree) 3 or higher, are difficult forks to pair. These forks correspond to complex saddles in f, such as monkey saddles. A single critical point pairing to these forks just reduces the degree of down-fork by 1, requiring complicated tracking of pairs. To simplify this, as seen in Fig. 2(d), these forks can be split into 2 forks ε apart. The upper down-fork retains 1 of the original down edges. The new down-fork connects with the old and takes the remaining down-edges. For even higher-order forks, the operation can be repeated on the lower down-fork.
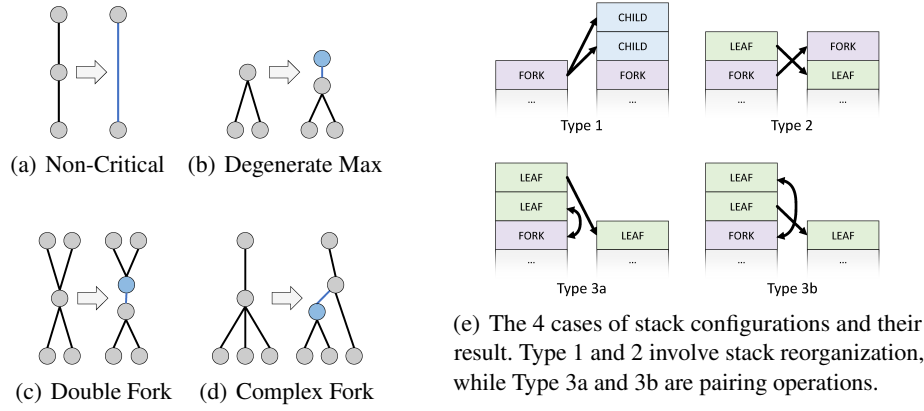
(a) Non-Critical   (b) Degenerate Max

(c) Double Fork   (d) Complex Fork

(e) The 4 cases of stack configurations and their result. Type 1 and 2 involve stack reorganization, while Type 3a and 3b are pairing operations.

Fig. 2. (a-d) Before pairing, the nodes of Reeb graph are conditioned considering 4 node configurations. New nodes and edges are shown in blue. (e) For non-essential fork pairing in the multipass algorithm, the 4 cases for stack processing are illustrated with their resulting configurations.

Beyond these requirements, the Reeb graph is assumed a single connected component. If not, each connected component can simply be extracted and processed individually. Finally, all nodes on the Reeb graph are assumed to have unique function values. If not, some processing order is arbitrary, and 0-persistence features may result.

4   Multipass Approach

The persistence diagram $\mathrm{Dg}_0(f)$ can be obtained by pairing the non-essential fork nodes of the Reeb graph. The extended persistence diagram $\mathrm{ExDg}_1(f)$ can be obtained by pairing of essential fork nodes. We demonstrate these 2 steps using Fig. 1 as an example.

4.1   Non-Essential Fork Pairing

Identifying the non-essential forks can be reduced to calculating join and split trees on the Reeb graph (see Fig. 1(c) and 1(d)), in our case, using Carr et al.'s approach [6]. Next, a stack-based algorithm, based upon branch decomposition [19], pairs critical points. The algorithm operates as a depth first search that seeks out simply connected forks (i.e., forks connected to 2 leaves) and recursively pairs and collapses the tree.

The algorithm processes the tree using a stack that is initially seeded with the root of the tree. At each iteration, 1 of 3 operation types occurs, as seen in Fig. 2(e). Operation Type 1 occurs when the top of the stack is a fork. In this case, the children of the fork are pushed onto the stack. Operation Type 2 occurs when the top of the stack is a leaf, but the next node is a fork. In this case, the leaf and fork have their orders swapped. Finally, operation Type 3 has 2 variants that occur when 2 leaf nodes sit atop the stack. In both variants, one leaf is paired with the fork, and the other leaf is pushed back onto the stack. The pairing occurs with the leaf that has a value *closer* to the value of the fork. The stack
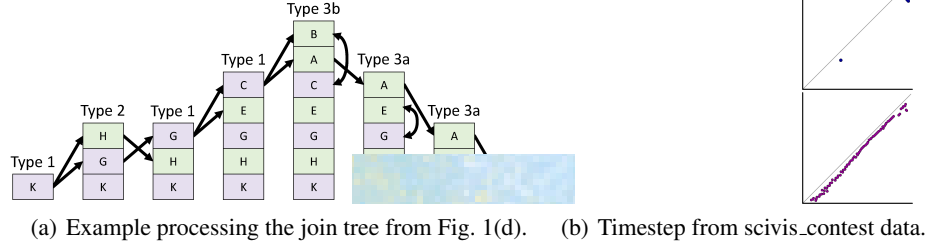
(a) Example processing the join tree from Fig. 1(d).   (b) Timestep from scivis_contest data.

Fig. 3. (a) An example pairing of the join tree from Fig. 1(d) shows the stack at each processing step, from left to right. (b) Timestep (066) from the scivis_contest data is shown with concentration mapped to color (left), along with Dg_0 (top) showing up-forks in blue and down-forks in red; and the $ExDg_1$ (bottom) showing cycles in purple.

is processed until only a single leaf node remains, the global minimum/maximum of the join/split trees, respectively. The algorithm operates identically on both join and split trees. Finally, the unpaired global minimum and maximum are paired.

Fig. 3(a) shows an example for the join tree in Fig. 1(d). Initially the root K is placed on the stack. A Type 1 operation pushes the children, G and H, onto the stack. Next, a Type 2 operation reorders the top of the stack. G, a down-fork, in now atop the stack, pushing its 2 children, E and C, onto the stack. Another Type 1 pushes C's children, A and B onto the stack. In the next 3 steps, a series of Type 3 operations occur. First B and C are paired, followed by E and G, and finally H and K. At the end, A, the global minimum, is the only point remaining on the stack. The assigned pairs, B/C, E/G, and H/K, appear in the $Dg_0(f)$ in Fig. 1(f), along with the split tree pairing, O/N, and the global min/max pairing, A/P.

4.2 Essential Forks Pairing

The remaining unpaired forks are essential forks, as seen in Fig. 1(e). We developed an algorithm from the high-level description of [4] to pair them. The essential fork pairing algorithm can be treated as join tree problem, processing forks one at a time. For a given up-fork, s, the node can be split into two temporary nodes, s_L and s_R. A join tree can be computed by sweeping the superlevel set. At each step of the sweep, the connected components are calculated. The pairing for a selected essential up-fork occurs at the down-fork that merges s_L and s_R into a single connected component.

Fig. 4 shows the sweeping process for the up-fork D. Initially (Fig. 4(a)), D is split into D_L and D_R, which are each part of separate connected components, denoted by color (Fig. 4(b)). As the join tree is swept past E (Fig. 4(c)), a new connected component is formed. In Fig. 4(d), F is added to the connected component of D_R. As the join tree is swept past G (Fig. 4(e)), the E and D_L connected components join. The process continues until Fig. 4(h), where 3 connected components exist. The purple and yellow components join at K (Fig. 4(i)). Finally at L (Fig. 4(j)), both D_L and D_R are part of the same connected component. This indicates that D pairs with L. Fig. 4(k-m) shows the

(a)   (b)   (c)   (k) *D* superlevel set (left) and join tree (right)

(d)   (e)   (f)   (l) *F* superlevel set (left) and join tree (right)

(g)   (h)   (i)   (j)   (m) *I* superlevel set (left) and join tree (right)
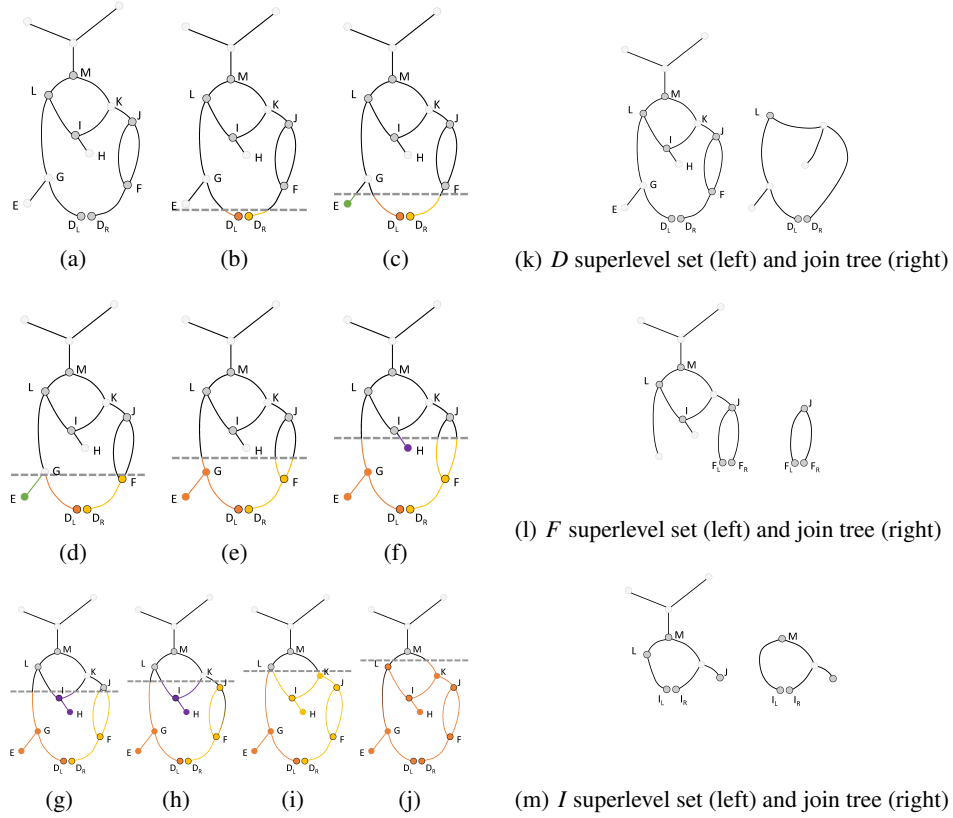
Fig. 4. Essential fork pairing in the multipass algorithm for the example Reeb graph from Fig. 1. (a-j) The join tree-based essential fork pairing for up-fork D. (a) D is initially split into D_L and D_R. (b-i) The colors indicate different connected components as the join tree is swept up the superlevel set. (j) The pairing is found when D_L and D_R are contained in the same connected component. (k-m) Each up-fork (D, F, and I, respectively) is split into 2 pieces and a join tree calculated from the superlevel set to find the partner.

superlevel sets and associated join trees for the up-forks D, F, and I. The pairing partner L/D, J/F, and M/I can all be seen in the $\mathrm{ExDg}_1(f)$ in Fig. 1(g).

5 Single-Pass Algorithm: Propagate and Pair

In the previous section, we showed that the critical point pairing problem could be broken down into a series of merge tree computations. For non-essential forks this was in the form of join and split trees, which are merge trees of the superlevel sets and sublevel sets, respectively. For essential saddles, it came in the form of a special join tree calculation for each essential up-fork. A natural question is whether these merge tree calculations can be combined into a single-pass operation, which is precisely what follows.

5.1   Basic Propagate and Pair

The Propagate and Pair algorithm operates by sweeping the Reeb graph from lowest to highest value. At each point, a list of unpaired points from the sublevel set is maintained. When a point is processed in the sweep, 2 possible operations occur on these lists: *propagate* and/or *pair*.

Propagate. The job of propagate is to push labels from unpaired nodes further up the unprocessed Reeb graph. 4 cases exist.

- For <u>local minima</u> a label for the current critical point is propagated upward. In the examples of Fig. 5(a) and 5(b), both A and B are propagated to C.
- For <u>local maxima</u> nothing needs to propagate.
- For <u>down-forks</u> all unpaired labels are propagated upwards. In the example of Fig. 5(c), the critical points B and C are paired, thus only A is propagated to D.
- For <u>up-forks</u> all unpaired labels are propagated upwards. Additional labels for the current up-fork are created and tagged with the specific branch of the fork that created them (in the examples with subscripts L and R). This tag is critical for closing essential cycles. In the example of Fig. 5(d), the labels A and D_L are propagated to G, and labels A and D_R are propagated to F.

Pair. The pairing operation searches the list of labels to determine an appropriate pairing partner from the sublevel set. The pairing operation only occurs for local maxima and down-forks.

- For <u>local maxima</u> the labels list is searched for the unpaired up-fork with the largest value. Those critical points are then paired. In Fig. 5(o), for local maximum O, the list is searched and N_L is determined to be the closest unpaired up-fork.
- For <u>down-forks</u> two possible cases exist, essential or non-essential, which can be differentiated by searching the available labels. First, the list is searched for the largest up-fork with both legs. Both legs indicate that the current down-fork closes a cycle with the associated up-fork. In the example, Fig. 5(m), the list of M is searched and labels I_L and I_R found. If no such up-fork exists, then the down-fork is non-essential. In this case, the highest valued local minimum is selected from the list. In the example of Fig. 5(c), no essential up-forks are found for C, and the largest local minimum, B is selected instead.

5.2   Virtual Edges for Propagate and Pair

The basic propagate and pair will fail in certain cases, such as in Fig. 6(a). The failure arises from the assumption that the superlevel set is the only thing needed to propagate labels. In this case, label information needs to be communicated between E and F, which are connected by the node D in the sublevel set. To resolve this communication issue, virtual edges are used. Virtual edges have 4 associated operations.

Virtual Edge Creation. Virtual edges are created on all up-fork operations. For example in Fig. 6(b), when processing B, the endpoints of the fork, E and F are connected with
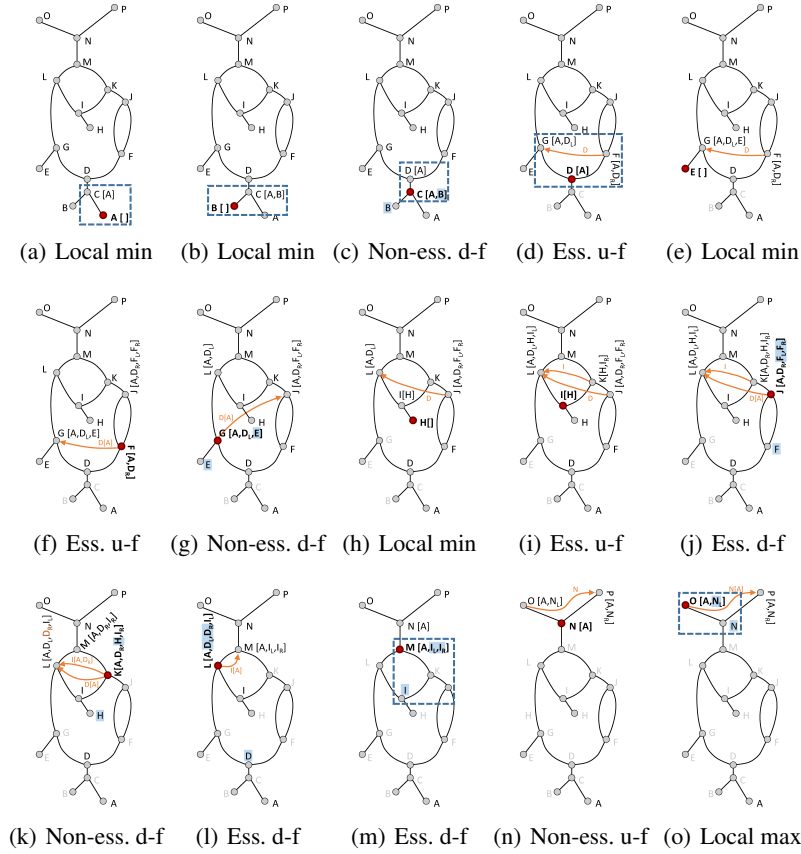
Fig. 5. Propagate and Pair algorithm on the Reeb graph from Fig. 1. At each step, the node being processed is in bold; propagated edges are shown in brackets; pairing is shown in blue; and virtual edges are shown in orange. (ess.: essential; non-ess.: non-essential; d-f: down-fork; u-f: up-fork)

virtual edge V_B. Similarly, in Fig. 6(c), when processing up-fork D, another virtual edge V_D is created connecting the endpoint, E and F.

Label Propagation. Propagating labels across virtual edges is similar to standard propagation with one additional condition. A label can only be propagated if its value is less than that of the up-fork that generated the virtual edge. In other words, for a given label X and a virtual edge V_Y, X is only propagated if $f(X) < f(Y)$. Looking at the example in Fig. 6(d), for the virtual edge V_B, only A is propagated because $f(A) < f(B)$. For the virtual edge V_D: A, B_L, and C are all propagated, since they all have values smaller than D.

Virtual Edge Merging. When processing down-forks, all incoming virtual edges need to be pairwise merged. Fig. 6(k) shows an example. When processing down-fork N, the
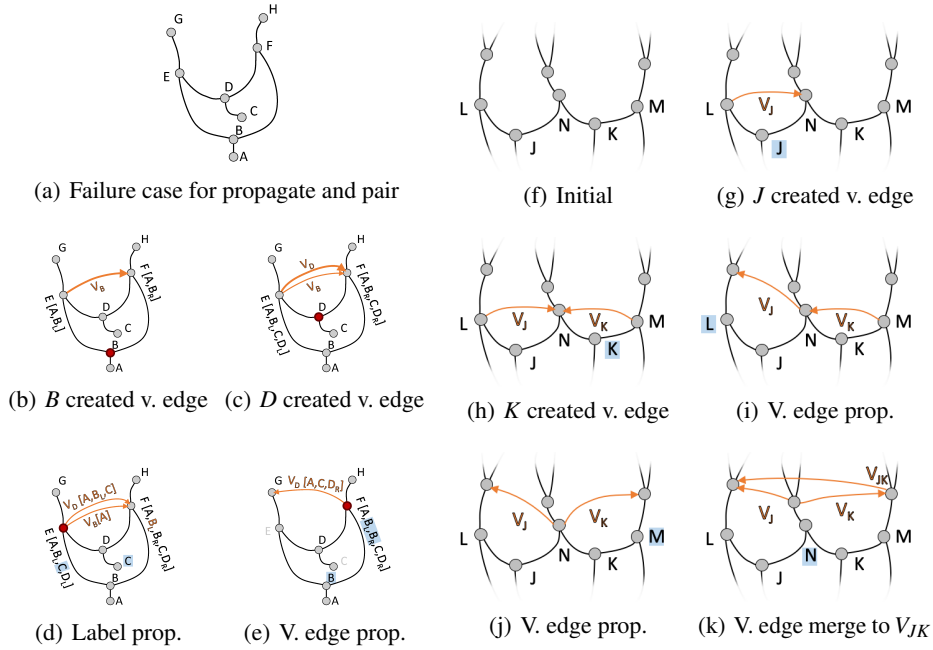
(a) Failure case for propagate and pair

(f) Initial

(g) *J* created v. edge

(b) *B* created v. edge   (c) *D* created v. edge

(h) *K* created v. edge

(i) V. edge prop.

(d) Label prop.

(e) V. edge prop.

(j) V. edge prop.

(k) V. edge merge to V_{JK}

Fig. 6. Example where the basic propagate and pair algorithm fails. (a-e) In this case, *B* and *F* should pair but do not. To overcome this limitation, (b-c) virtual edges are created as up-forks are processed. (d) Labels can then be propagated across virtual edges. (e) The virtual edges themselves are propagated and redundant edges removed. (f-k) An example requiring virtual edge merging. (g-h) Virtual edges are created. (i-j) Virtual edges are propagated. (k) At the down-fork *N*, virtual edges V_J and V_K are propagated and merged into V_{JK}.

virtual edges V_J and V_K are merged into a new virtual edge V_{JK}. For the purpose of label propagation, the virtual edge uses its minimum saddle, in this case *J*.

Virtual Edge Propagation. Finally, virtual edges themselves need to be propagated. For up-forks, all virtual edges are propagated up to both neighboring nodes. In the case of down-forks, all virtual edges are similarly propagated, as we see in Fig. 6(e). During the virtual edge propagation phase, redundant virtual edges can also be culled. For example, the virtual edge V_D is a superset of V_B. Therefore, V_B can be discarded. The necessity of the virtual edge process can also be seen in Fig. 5. In Figures 5(i)-5(l), the pairing of *L* with D is only possible because of the virtual edge created by *I* in Fig. 5(i).

6   Evaluation

We have implemented the described algorithms using Java. Performance reported in Table 1 was calculated on a 2017 MacBook Pro, 3.1 Ghz i5 CPU, 8 GB RAM.

We investigate the performance of the algorithms using the following:

– Synthetic split trees, join trees, and Reeb graphs generated by a Python script. Given a positive integer n, where $n = \{100, 500, 1000, 3000, 5000\}$, the script creates a fork G_1 consisting of a node with valency 3 and 3 nodes with valency 1 linked to the 3-valence node. At each iteration $i < n$, another fork is generated, and 1 or 2 of its 1 valency nodes are glued to the nodes in G_{i-1} with valency 1. Constraining the gluing to a single node at each iteration results in a split tree.

– Reeb graphs calculated on publicly available meshes in Figures 8 and meshes provided by AIM@SHAPE Shape Repository. Reeb graphs were extracted using our own Reeb graph implementation in C++.

– Time-series of 120 Mapper graphs taken from the 2016 SciVis Contest[3], a large time-varying multi-run particle simulation, in Fig. 3(b). Our evaluation took one realization, smoothing length 0.44, run 50, and calculated the Mapper graphs for all 120 time-steps using the variable *concentration*. Our video, available at `https://youtu.be/AcJX4GdzBZY`, shows the entire sequence. The Mapper graphs were generated using a Python script that follows the standard Mapper algorithm [23].

Overview of Results. The performance for the algorithms can be seen in Table 1. These values were obtained by running the test 1000 times and storing the average compute time. The persistence diagrams of both the single-pass and multipass algorithms were compared in order to verify correctness. For most cases, the single-pass approach outperformed the multipass approach. The exceptions being the random split tree, random graph, and SciVis contest data, each of which we will discuss.

---

[3] `https://www.uni-kl.de/sciviscontest/`

Table 1. Performance for all datasets tested. Bold indicates the faster algorithm.

Data	Figure	Mesh		Reeb Graph Nodes		Cycles	Multipass Time (ms)	Single-pass Time (ms)
		Vertices	Faces	Initial	Cond.			
random_tree_100				401	204	0	2.45e-02	2.71e-02 (split) **9.06e-03** (join)
random_tree_500				2001	1004	0	0.13	0.18 (split) **4.90e-02** (join)
random_tree_1000				4001	2004	0	0.42	0.30 (split) **0.11** (join)
random_tree_3000				12001	6004	0	1.10	1.98 (split) **0.39** (join)
random_tree_5000				20001	10004	0	2.11	3.39 (split) **0.75** (join)
random_graph_100				401	112	46	1.90e-02	**1.76e-02**
random_graph_500				2001	542	231	**0.48**	0.57
random_graph_1000				4001	1010	497	**0.55**	0.59
random_graph_3000				12001	3014	1495	**1.71**	1.91
random_graph_5000				20001	5204	2400	**14.35**	24.45
4_torus	8(d)	10401	20814	23	10	4	2.06e-03	**1.47e-03**
buddah	8(c)	10098	20216	33	14	6	1.61e-03	**1.16e-03**
david	8(e)	26138	52284	8	8	3	7.82e-04	**4.17e-04**
double_torus	8(a)	3070	6144	13	6	2	5.29e-04	**2.80e-04**
female	8(b)	8410	16816	15	8	0	7.82e-04	**3.45e-04**
flower	8(h)	4000	8256	132	132	65	2.80e-02	**2.43e-02**
greek	8(f)	39994	80000	23	10	4	8.62e-04	**4.81e-04**
topology	8(g)	6616	13280	28	28	13	4.34e-03	**4.02e-03**
scivis_contest	3(b)	194k (avg)	—	117 (avg)	178.2 (avg)	81.3 (avg)	**3.82** (total)	4.18 (total)

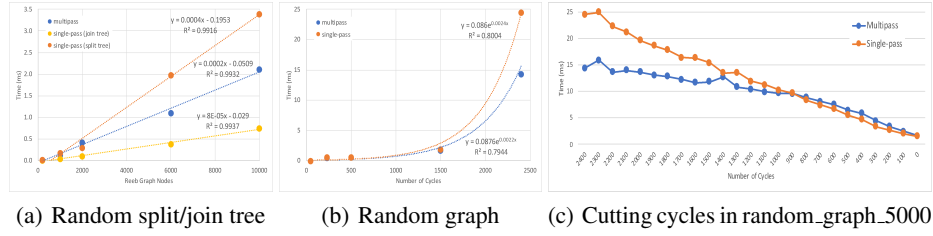(a) Random split/join tree   (b) Random graph   (c) Cutting cycles in random_graph_5000

Fig. 7. Plots of the compute time for various input sizes to (a) the random split/join tree and (b) the random graph for Table 1. (c) Performance results when cutting cycles in the random_graph_5000. As more cycles that are cut, the single-pass algorithm begins to outperform the multipass variant.

Random Split Tree vs. Join Tree. We compared the exact same tree structures as split trees and join trees by negating the function value of the input tree. The performance observed in Table 1 and Fig. 7(a) shows that the join tree performs significantly better than the split tree. The explanation for this is quite simple. The join tree consists of exclusively down-forks, while the split tree consists of exclusively up-forks. Since only up-forks generate virtual edges, the split tree created and processed many virtual edges, while the join tree has none. In fact, split trees represent one worst case by generating many unneeded virtual edges. From a practical standpoint, the algorithm can avoid situations like this by switching sweep directions (i.e. top-to-bottom), when the number of up-forks is significantly larger than the number of down-forks.

Random Graph. We next investigate the performance of randomly generated Reeb graphs, shown in Table 1 and Fig. 7(b). These Reeb graphs consist predominantly of cycles. This represents another type of worst case, since many up-forks generate virtual edges, which are then merged into even more virtual edges at the down-forks. To verify this, we ran an experiment, as seen in Fig. 7(c), that randomly cuts n cycles in the starting Reeb graph random_graph_5000 containing 2400 cycles. The break even was about 900 cycles (about 25% essential and 75% non-essential forks).

SciVis Contest Data. The SciVis contest data was "cycle heavy" as can be seen in the persistence diagram of Fig. 3(b). Given the random graph analysis, it is unsurprising that the performance of the single-pass approach was lower than the multipass approach.

7   Discussion & Conclusion

Pairing critical points is a key part of the TDA pipeline—the Reeb graphs capture complex structure, but direct representation is impractical. Critical point pairing enables a compact visual representation in the form of a persistence diagram. The value of representing a dataset with the persistence diagram is the simplicity and efficiency. Persistence diagrams avoid the occlusions problems of normal 3D datasets (e.g., the internal structure of Fig. 3(b)), and they avoid the potential confusion of direct representation of
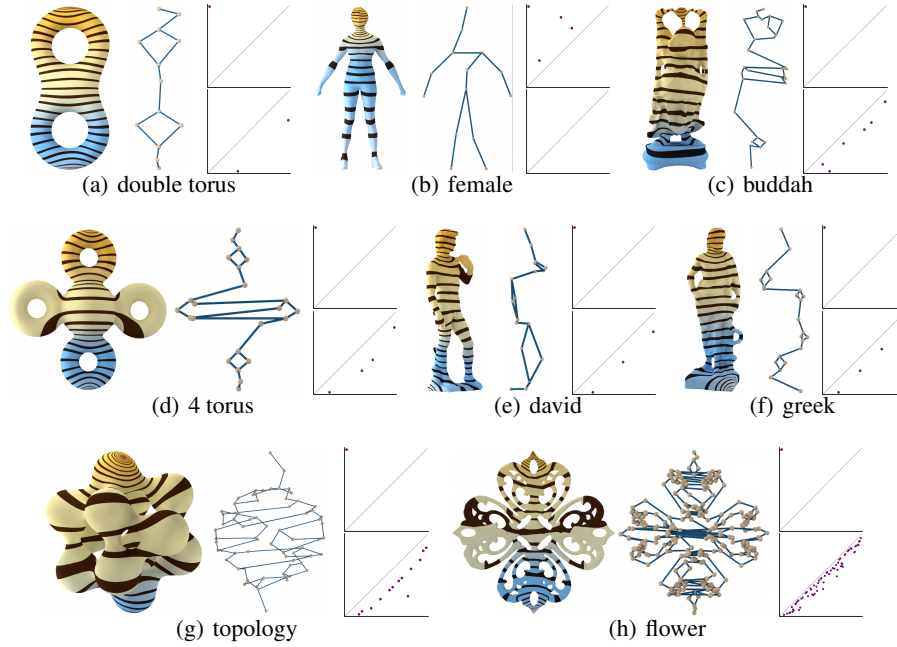
Fig. 8. The meshes colored by the scalar value (left), Reeb graphs (middle), Dg_0 with up-forks in blue and down-forks in red (top), and $ExDg_1$ cycles in purple (bottom) are shown for evaluation.

the Reeb graph (e.g., the Reeb graph of Fig. 8(h)). In addition, they provide sharp visual cue for time-varying data (see our video).

Our results showed that although the single-pass algorithm tended to outperform the multipass algorithm, there was no clear winner. We point out some advantages and disadvantages for each. The multipass algorithm has an advantage in simplicity of implementation. Once the merge tree and branch decomposition are implemented, the only necessity is repeated calls to those algorithms. This approach also has a potential advantage for (limited) parallelism. First, processing join and split trees in parallel, then all essential up-forks. The single-pass algorithm showed a slight edge in performance, particularly for data with a balance between essential and non-essential forks. The other significant advantage of the approach is that it is in fact a single-pass approach, only visiting critical points once. This is useful for streaming or time-varying data, where the critical points arrive in order, but analysis cannot wait for the entire data to arrive.

References

1. Agarwal, P.K., Edelsbrunner, H., Harer, J., Wang, Y.: Extreme elevation on a 2-manifold. Discrete & Computational Geometry **36**(4), 553–572 (2006)

2. Attene, M., Biasotti, S., Spagnuolo, M.: Shape understanding by contour-driven retiling. The Visual Computer **19**(2), 127–138 (2003)
3. Bajaj, C.L., Pascucci, V., Schikore, D.R.: The contour spectrum. In: Proceedings of the 8th IEEE Visualization. pp. 167–ff (1997)
4. Bauer, U., Ge, X., Wang, Y.: Measuring distance between reeb graphs. In: Symposium on Computational Geometry. p. 464 (2014)
5. Boyell, R.L., Ruston, H.: Hybrid techniques for real-time radar simulation. In: Proceedings of the November 12-14, 1963, fall joint computer conference. pp. 445–458 (1963)
6. Carr, H., Snoeyink, J., Axen, U.: Computing contour trees in all dimensions. Computational Geometry: Theory and Applications **24**(2), 75–94 (2003)
7. Carr, H., Snoeyink, J., van de Panne, M.: Simplifying flexible isosurfaces using local geometric measures. Proceedings 15th IEEE Visualization pp. 497–504 (2004)
8. Cole-McLaughlin, K., Edelsbrunner, H., Harer, J., Natarajan, V., Pascucci, V.: Loops in reeb graphs of 2-manifolds. In: Symposium on Computational Geometry. pp. 344–350 (2003)
9. Doraiswamy, H., Natarajan, V.: Efficient output-sensitive construction of reeb graphs. In: International Symposium on Algorithms and Computation. pp. 556–567. Springer (2008)
10. Doraiswamy, H., Natarajan, V.: Efficient algorithms for computing reeb graphs. Computational Geometry **42**(6), 606–616 (2009)
11. Doraiswamy, H., Natarajan, V.: Computing reeb graphs as a union of contour trees. IEEE Transactions on Visualization and Computer Graphics **19**(2), 249–262 (2013)
12. Edelsbrunner, H., Harer, J., Mascarenhas, A., Pascucci, V.: Time-varying reeb graphs for continuous space-time data. In: Symposium on Computational Geometry. pp. 366–372 (2004)
13. Edelsbrunner, H., Letscher, D., Zomorodian, A.: Topological persistence and simplification. In: Symposium on Foundations of Computer Science. pp. 454–463 (2000)
14. Harvey, W., Wang, Y., Wenger, R.: A randomized O (m log m) time algorithm for computing Reeb graphs of arbitrary simplicial complexes. Symp. on Comp. Geo. pp. 267–276 (2010)
15. Hilaga, M., Shinagawa, Y.: Topology matching for fully automatic similarity estimation of 3D shapes. SIGGRAPH pp. 203–212 (2001)
16. Kweon, I.S., Kanade, T.: Extracting topographic terrain features from elevation maps. CVGIP: image understanding **59**(2), 171–182 (1994)
17. Munch, E.: A users guide to topological data analysis. J. Learn. Analytics **4**(2), 47–61 (2017)
18. Parsa, S.: A deterministic $O(m \log m)$ time algorithm for the Reeb graph. In: ACM Sympos. Comput. Geom. (SoCG). pp. 269–276 (2012)
19. Pascucci, V., Cole-McLaughlin, K., Scorzelli, G.: Multi-resolution computation and presentation of contour trees. In: IASTED Conf. Vis., Img., and Img. Proc. pp. 452–290 (2004)
20. Pascucci, V., Scorzelli, G., Bremer, P.T., Mascarenhas, A.: Robust on-line computation of Reeb graphs: Simplicity and speed. ACM Transactions on Graphics **26**(3), 58.1–58.9 (2007)
21. Reeb, G.: Sur les points singuliers dune forme de pfaff completement intgrable ou dune fonction numrique. CR Acad. Sci. Paris **222**, 847–849 (1946)
22. Rosen, P., et al.: Using contour trees in the analysis and visualization of radio astronomy data cubes. In: TopoInVis (2019)
23. Singh, G., Mémoli, F., Carlsson, G.E.: Topological methods for the analysis of high dimensional data sets and 3d object recognition. In: Eurographics SPBG. pp. 91–100 (2007)
24. Takahashi, S., Takeshima, Y., Fujishiro, I.: Topological volume skeletonization and its application to transfer function design. Graphical Models **66**(1), 24–49 (2004)
25. Tierny, J., Vandeborre, J.P., Daoudi, M.: Partial 3D shape retrieval by Reeb pattern unfolding. Computer Graphics Forum **28**(1), 41–55 (2009)