Efficient Algorithms and Applications in Topological Data Analysis

Junyi Tu
University of South Florida, junyitu@gmail.com

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons, and the Mathematics Commons

Scholar Commons Citation
Tu, Junyi, "Efficient Algorithms and Applications in Topological Data Analysis" (2016). Graduate Theses and Dissertations.
https://scholarcommons.usf.edu/etd/8083

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Efficient Algorithms and Applications in Topological Data Analysis

by

Junyi Tu

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Paul Rosen, Ph.D.
Les Piegl, Ph.D.
Yicheng Tu, Ph.D.
Zhuo Lu, Ph.D.
Dmytro Savchuk, Ph.D.

Date of Approval:
November 7, 2019

Keywords: Reeb Graphs, Contour Trees, Parallel Computing, Persistent Pairing, Image Enhancement

Copyright © 2019, Junyi Tu
Dedication

For Wenli and Joshua, who taught me yet another language.
Acknowledgments

I am deeply indebted to my adviser, Dr. Paul Rosen, without whom I would not have a chance to earn a Ph.D in Computer Science and Engineering. Especially at the same time I am starting a family in a foreign country. Without the support from my adviser, I can not survive this adventure. The training in CSE is very different from my previous training in mathematics, it really broadens my horizon to another dimension in computers invented less than one hundred years ago. I would also like to extend my deepest gratitude to my committee members for their encouragement and advice: Dr. Yicheng Tu, Dr. Les Piegl, Dr. Zhuo Lu and Dr. Dmytro Savchuk. Special thanks go to my defense chairperson, Dr. Alon Friedman, who also provided feedback to improve this dissertation.

I would like to thank several other faculty members from whom I have learned a lot in programming languages, machine learning and computer security. Those professors include Dr. Dmitry B. Goldgof, Dr. Jarred Ligatti, Dr. Yu Sun, Dr. Hao Zheng, Dr. Larry Hall and many others. I owe a lot to Dr. Yuncheng You for his encouragement and support. I am grateful to my internship hosts, Dr. Schanen Michel and Dr. Kibaek Kim in Argonne National Lab, they gave me a rare chance to see the research involving both mathematics and computer science. My classmates and friends also help me a lot, I especially want to thank Bingxiong Lin, Jie Zhang, Lei Wu, Xiaofan Xu, Yuqi Huang, Zhaolin Jiang, Nan Zhao, Jun Cheng, Yongqiang Huang, Ran Rui, Renhao Liu, Jiayi Wang, Xiaoshan Wang, Mark Hoel, ATM Golam Bari, Cagri Cetin, Jean-Baptiste Subils, Chengcheng Mou and many others. I must also thank my great labmates: Zachariah J. Beasley, Tanmay Kotha, Ghulam Jilani Quadri, and Zhila Nouri. Last but not least, I want to thank the staff of the CSE main office, especially Laura and Gabriela, for their help and the entire department for their warmth and kindness.
# Table of Contents

List of Tables iv  
List of Figures v  
Abstract viii  
Chapter 1: Introduction 1  
  1.1 Motivation 1  
  1.2 Contribution 4  
Chapter 2: Background 7  
  2.1 Critical Points 7  
  2.2 Contour Tree 7  
  2.3 Reeb Graph 10  
  2.4 Mapper 11  
  2.5 Critical Point Pairing 13  
    2.5.1 Persistent Homology and Persistence Diagram 13  
Chapter 3: Parallel Computation of Merge Tree 17  
  3.1 Introduction 17  
  3.2 Conventional Merge Tree Construction 19  
  3.3 Parallel Merge Tree Construction 21  
    3.3.1 Phase 1: Coloring 21  
    3.3.2 Phase 2: Potential Critical Point Extraction 22  
    3.3.3 Phase 3: Saddle Sorting 23  
    3.3.4 Phase 4: Subtree Building and Merge Propagation 24  
  3.4 Extension to 3D and Higher Dimensions 24  
  3.5 OpenCL Implementation 25  
  3.6 Experiments 25  
    3.6.1 Random Field Tests 25  
    3.6.2 Contour Trees in Radio Astronomy Data 28  
  3.7 Conclusions 30
Chapter 4: Critical Point Pairing in Reeb Graphs

4.1 Introduction
4.2 Reeb Graph
   4.2.1 Persistence Diagram of Reeb Graph
      4.2.1.1 Branching Features of Reeb Graph
      4.2.1.2 Cycle Features of Reeb Graph
   4.2.2 Related Work
4.3 Conditioning the Graph
4.4 Multipass Approach
   4.5.1 Non-Essential Fork Pairing
   4.5.2 Essential Fork Pairing
4.5 Single-Pass Algorithm: Propagate and Pair
   4.6.1 Basic Propagate and Pair
   4.6.2 Virtual Edges for Propagate and Pair
4.6 Evaluation
4.7 Conclusions

Chapter 5: Contrast and Brightness Enhancement of Color Image

5.1 Introduction
   5.1.1 Computing the Contour Tree
5.2 Capturing the Topology of an Image
   5.2.1 Feature Subtree Extraction
   5.2.2 The Contour Tree of Color Images
5.3 Image Processing via Contour Trees
   5.3.1 Visualizing the Contour Tree
   5.3.2 Subtree Selection as Image Segmentation
   5.3.3 Subtree Modification as Image Editing
5.4 Examples
5.5 Conclusions

Chapter 6: Application of Mapper in Point Cloud-based 3D Printed Objects

6.1 Introduction
6.2 Mapper and Persistent Homology
   6.2.1 Mapper
   6.2.2 Persistent Homology
   6.2.3 Link Between Mapper and Persistent Homology
6.3 The Topology of 3D Printing
   6.3.1 Visualization
6.4 Results
   6.4.1 Original Model
   6.4.2 Error Corrected Model
   6.4.3 Runtime Performance
6.5 Conclusions
Chapter 7: Conclusion and Future Work

7.1 Summary
7.2 Vision

References

Appendix A: Copyright Permissions
List of Tables

Table 4.1  Performance for all datasets tested. 54
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Topological Data Analysis Pipeline.</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Critical points of a scalar function.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Contour tree calculation on a scalar field.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Reeb graph on a manifold with a function.</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Mapper on a point cloud model.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Example of persistence diagram.</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Three existing strategies used to parallelize merge tree construction.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Example of 2D scalar field used to describe our parallelization of merge tree construction.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Illustration of conventional merge tree construction.</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Illustration of the four phases of parallel merge tree construction.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Example of noisy scalar fields used for performance testing.</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Charts of the performance of our parallel merge tree algorithm on random fields.</td>
<td>27</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Visualization of Radio Astronomy Data.</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Performance on Radio Astronomy Data.</td>
<td>30</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Reeb graph and persistence diagram.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Cases for Reeb graph conditioning.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Example of multipass critical point paring.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Illustration of the four case for non-essential fork pairing in the multipass algorithm.</td>
<td>40</td>
</tr>
</tbody>
</table>
Figure 4.5 Illustration of non-essential fork pairing in the multipass algorithm.  
Figure 4.6 The join tree-based essential fork pairing for up-fork D.  
Figure 4.7 Essential fork pairing in the multipass algorithm for the example Reeb graph from Figure 4.3.  
Figure 4.8 Propagate and Pair algorithm on the example Reeb graph from Figure 4.3.  
Figure 4.9 Continuation of Figure 4.8.  
Figure 4.10 An example case of where the basic propagate and pair algorithm fails.  
Figure 4.11 An example requiring virtual edge merging.  
Figure 4.12 Persistence diagrams for random trees and random graphs.  
Figure 4.13 Meshes and Reeb graphs used in evaluation.  
Figure 4.14 Continuation of Figure 4.13.  
Figure 4.15 Performance on random split/join trees and graphs.  
Figure 4.16 Persistence diagram of SCIVIS contest data.  
Figure 4.17 Performance results when cutting cycles in the random_graph_5000.  
Figure 5.1 Contour tree on a terrain.  
Figure 5.2 Contour tree and its decomposition.  
Figure 5.3 Interfaces used for selecting topological features.  
Figure 5.4 Example of segmentation based on persistence diagram.  
Figure 5.5 Four functions based on subtree editing.  
Figure 5.6 Illustration of four functions on scalar field  
Figure 5.7 Evaluation on Florala dataset.  
Figure 5.8 Evaluation on Brain dataset.  
Figure 5.9 Evaluation on Lenna Grayscale dataset  
Figure 5.10 Evaluation on Notre Dame dataset.  
Figure 5.11 Evaluation on Lenna Color dataset.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.12</td>
<td>Evaluation on Swan dataset.</td>
<td>76</td>
</tr>
<tr>
<td>6.1</td>
<td>Multilayers filled/Empty space topology.</td>
<td>78</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of Mapper on a mesh.</td>
<td>79</td>
</tr>
<tr>
<td>6.3</td>
<td>Example of Mapper on the empty space of a mesh.</td>
<td>80</td>
</tr>
<tr>
<td>6.4</td>
<td>Example of persistent homology</td>
<td>81</td>
</tr>
<tr>
<td>6.5</td>
<td>Our software with the Stanford Dragon dataset.</td>
<td>83</td>
</tr>
<tr>
<td>6.6</td>
<td>Results of Dragon dataset.</td>
<td>84</td>
</tr>
<tr>
<td>6.7</td>
<td>Error Corrected Results of Dragon dataset.</td>
<td>86</td>
</tr>
<tr>
<td>6.8</td>
<td>Performance Evaluation.</td>
<td>88</td>
</tr>
</tbody>
</table>
Abstract

Topological Data Analysis (TDA) is a new and fast growing research field developed over last two decades. TDA finds many applications in computer vision, computer graphics, scientific visualization, molecular biology, and material science, to name a few. In this dissertation, we make algorithmic and application contributions to three data structures in TDA: contour trees, Reeb graphs, and Mapper. From the algorithmic perspective, we design a parallel algorithm for contour tree construction and implement it in OpenCL. We also design and implement critical point pairing algorithms to compute persistence diagrams directly from contour trees, Reeb graphs, and Mapper. In terms of applications, we apply TDA in the design and implementation of an image enhancement application using contour trees. Lastly, we introduce an application of Mapper and persistent homology in model quality assessment for 3D printing.
Chapter 1: Introduction

1.1 Motivation

Geometric shapes are ubiquitous in many datasets from computer graphics, computer
aided design, computer vision, and data visualization. These research fields utilize count-
less problems in computational geometry, such as convex hull, Voronoi diagrams, Delaunay
triangulation, polygon partitioning, and geometric searching. Many efficient and effective
algorithms to solve these problems are presented in the classic textbook by Franco Preparata
and Michael Shamos [61].

Topology studies the properties of a geometric object that are preserved under continuous
deformation, such as stretching, twisting, and bending, but not tearing. One common ex-
ample of continuous deformation is transforming a coffee mug into a donut. Each is a single
component with one hole, i.e., the mug handle and donut hole. In other words, topologists
do not see a difference between the shape of a coffee mug and a donut. An early precuror to
modern topology was the well-known “Seven Bridges of Königsberg”, which Leonhard Eu-
ler (1707–1783) proved had no solution and inspired the polyhedron formula $V - E + F = 2$,
where $V$, $E$, and $F$ denoting the number of vertices, edges, and faces of the polyhedron,
respectively. Topology can also reveal the intrinsic secrets of exotic matter—the 2016 Nobel
Prize in Physics was awarded to David J. Thouless, F. Duncan M. Haldane, and J. Michael
Kosterlitz, “for theoretical discoveries of topological phase transitions and topological phases
of matter” [37].
Unfortunately, combining topological invariants with geometry to solve real world problems is nontrivial. Approximately two decades ago Topological Data Analysis (TDA) began to bridge this gap, most notably with the work on persistent homology by Herbert Edelsbrunner [35], Afra Zomorodian [90], and Gunnar Carlsson [14]. The classic workflow of TDA is as follows: the input of TDA pipeline is a point cloud dataset; then, a filtration, a nested sequence of complexes, such as the alpha complex, is built on the point cloud; next, the homology of each complex in the filtration is used to calculate the persistence module; lastly, the matrix reduction algorithm produces the persistence diagram, which characterizes the topological features by defining a scale parameter that tracks the birth and death, i.e., appearance and disappearance, of features continuously over all spatial resolutions. The persistence diagram is often visualized as a scatterplot of the same name, i.e., a persistence diagram, which indexes topological features by their “life-time” in the filtration.

Visualization has been used to analyze and understand data for decades [22, 81], and TDA combines both data analysis and visualization seamlessly. Topology-based techniques for analyzing data are becoming increasingly popular due in large part to their robustness and their applicability to a wide variety of datasets and scientific domains, from cancer research [54] to sports analytics [4], to collaboration [7, 19] and brain networks [20, 25], among others [15, 46, 47, 51, 77, 53]. Many open-source libraries and software packages have been developed for the computation of persistent homology, such as Dionysus [48], JavaPlex [1], PHAT (Persistent Homology Algorithm Toolkit), and DIPHA (Distributed Persistent Homology Algorithm) [9], among many others. [56] provides a detailed analysis of software packages and tutorial for the computation of persistent homology on point cloud datasets.

Another branch of TDA uses other data structures, including contour trees [12], Reeb graphs [63], and Mapper [70], to compute persistence and persistence-like features under special conditions or on other types of data, e.g., manifolds [13]. These tools are useful for
computing persistence on data that do not easily fit in the persistent homology pipeline, such as a function applied to a 2-manifold, i.e., the Buddha mesh, or a simple domain, i.e., the grayscale image, in Figure 1.1.

The Reeb graph [63] was first introduced in the mid-19th century. A more detailed introduction to the Reeb graph is included in the Background chapter, but in short, Reeb
Graph encodes the evolution of the connectivity of the level sets induced by a scalar function defined on a data domain by sweeping from negative infinity to positive infinity and tracking the birth and death of the connected components of the level sets. A non-looping Reeb graph is called the \textit{contour tree} [12], and an approximation for the Reeb graph can be computed using \textit{Mapper} [70].

One challenge with using Reeb graphs to directly analyze data is that the graphs they produce are frequently too large or complex to directly visualize, therefore requiring further abstraction. 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 pair the critical points in the Reeb graph [2].

Reeb graphs, contour trees, and Mapper have found many applications in computer graphics, computer vision, and scientific visualization. They have been successfully used in feature detection [73], data reduction and simplification [17, 68], image processing [45], shape understanding [5], visualization of isosurfaces [6], analysis and visualization tool of radio astronomy data [66], etc. One widely used tool for this type of computation is the Topology ToolKit (TTK) [75], which primarily computes contour trees and Reeb graphs on triangulated mesh data in 2D and 3D. These algorithms are the main focus of this dissertation (see Figure 1.1).

1.2 Contribution

When broadly considering TDA, we address two main areas of need. The first is in efficient and parallel algorithms for computing TDA data structures, particularly for large data. TDA calculations tend to be either expensive to compute, e.g., persistent homology is $O(n^3)$, or extremely difficult to parallelize, e.g., contour trees require a partial ordering with complicated communication patterns. The second challenge is considering the utility
of TDA in domain applications. In these domains it may be nontrivial to map the domain problem to a TDA data structure, therefore requiring redesigning the TDA data structure.

In this dissertation, we address both the algorithmic and application challenges of TDA using three data structures: contour trees, Reeb graphs, and Mapper.

- First, we design and implement an efficient approach to parallel computation of a merge tree on a graphical processing unit (GPU). The merge tree represents an essential component and major bottleneck to building a contour tree or Reeb graph.

- Second, we design and implement two algorithms for efficient computation of the critical point pairing in Reeb graphs, contour trees, and Mapper.

- Third, we adapt the contour tree to the application of color image enhancement. In this work we generalize topology-based denoising to introduce new topology-based transfer functions, including contrast enhancement, brightness enhancement, and gamma correction, that can be applied to contour trees to interactively enhance color images.

- Lastly, we study an application of TDA in assessing the quality of point cloud-based 3D printed objects. In particular, we present an approach that uses a hybrid of Mapper and persistent homology to identify anomalies in point cloud models that may produce printer errors.

In summary, this dissertation demonstrates that the algorithms of TDA can be efficiently implemented and used in a wide variety of applications to solve real world problems. The dissertation is organized as follows. In Chapter 3, we present the parallel computation of merge tree. Chapter 4 is devoted to the algorithms of critical point pairing in Reeb graphs, one multi-pass and one single-pass algorithm. In Chapter 5, we show the application of contour tree and persistence diagrams in color images enhancement. Chapter 6 presents an application of Mapper and persistent homology in assessing the quality of models for point
cloud-based 3D printed objects. Finally, we conclude our work and present future directions in Chapter 7.
Chapter 2: Background

2.1 Critical Points

The topological data structures that this dissertation focuses on, specifically contour trees, Reeb graphs, and Mapper, are all concerned with capturing three geometric properties of data, local minima, local maxima, and saddle points (Figure 2.1). These being the three cases when the local function tangent is 0. These critical points are useful for tasks, such as hierarchical visualization [6, 17], segmentation [28, 72], or tracing structures [83] in scalar field data.

![Figure 2.1. Critical points of a scalar function. Functions representing critical points: (a) local minimum, (b) local maximum, and (c) a saddle point.](image)

2.2 Contour Tree

Contour tree was first introduced by Boyell and Ruston [12] in 1963, who named it the “enclosure tree” of contour lines for the height of terrain. After an efficient serial algorithm
was introduced by Carr et al. [16], the contour tree became a popular tool in scientific visualization due to its ability to capture the topological structure of scalar fields. Contour trees have been used in many applications, such as isoline extraction in geometric data by van Krevald et al. [83], which provided the first formal algorithm, fast extraction of isosurfaces and interactive data exploration [58], volume rendering [85], uncertain terrains [88], and noise removal that does not negatively impact important features in the data [68].

The contour tree (see Figure 2.2(c)) is obtained by contracting the connected components of the levelsets (i.e., the isocontours) of a function defined on a simple domain (i.e., a domain without any non-trivial homology) to a point and tracking the evolution of those levelsets. The construction of the contour tree can be decomposed into two parts, the join tree (see Figure 2.2(e)) and split tree (see Figure 2.2(g)), both of which are obtained using a merge tree that sweeps the function from bottom-to-top and top-to-bottom, respectively. As the merge tree sweeps the function, it tracks connected components in what are known as augmented versions of the trees. The final trees are found by removing non-critical points (those with one down edge and one up edge).

Figure 2.2 shows an example of contour tree construction. In Figure 2.2(a), a 4-by-4 scalar field is represented by color from black (lower value) and white (higher value). To construct the contour tree, the augmented join tree (see Figure 2.2(d)) and augmented split tree (see Figure 2.2(f)) are found and combined into an augmented contour tree (see Figure 2.2(b)). When non-critical points are removed from those trees, the join tree (see Figure 2.2(e)), split tree (see Figure 2.2(g)) and contour tree (see Figure 2.2(c)) are found.

The time complexity of Carr et al.’s algorithm is $O(n \log n)$, primarily coming from a global sorting of points required to compute the augmented join and split trees. Furthermore, the algorithm requires a complete global ordering of data to guarantee the correct output. However, it can be shown, as we do in Chapter 3, that partial ordering is sufficient for most aspects of the algorithm, enabling the opportunity for parallelism.
Figure 2.2. Contour tree calculation on a scalar field. The (a) scalar field has its (b) augmented contour tree calculated by combining the (d) augmented join tree and (f) augmented split tree. The critical points of the augmented trees are used to generate the (c) contour tree, (e) join tree, and (g) split tree.
2.3 Reeb Graph

Reeb graph is named after French mathematician Georges Reeb (1920–1993), who proposed the graph as a tool in Morse theory around 1946 [63]. The Reeb graph is obtained similarly to the contour tree by contracting the connected components of the levelsets of a function. In this case, the function is defined on a manifold with non-trivial homology (e.g., holes/tunnels). In fact, the contour tree is an acyclic Reeb graph. It provides a skeletonized summarization of both geometrical shape and its relationship with the scalar function. Hence, it is an instrumental data structure to encode the topological properties of geometric objects. Figure 2.3 shows an example Reeb graph captured from a manifold with a scalar function defined on it.

![Manifold with a scalar function](image1)

![Reeb graph](image2)

Figure 2.3. Reeb graph on a manifold with a function. The (a) manifold with a scalar function is processed into (b) a Reeb graph that encodes both the geometric shape and its relationship to the scalar function.
Reeb graphs and contour trees have found numerous applications in graphics and visualization including data skeletonization [38], locus cut [26], data abstraction [52], retrieving topological information from point data, such as homology group computation [27, 21], volume rendering [86], and terrain applications [10, 40].

The first algorithm to compute Reeb graph on a triangulated surface was presented by Shinagawa and Kunii [69], with time complexity $O(n^2)$, where $n$ is the number of triangles in the mesh. The efficient computation of Reeb graphs has been an active research topic for last two decades. Cole-McLaughlin et al. [24] improved the performance to $O(n \log n)$. Pascucci et al. [60] presented an online method to compute Reeb graphs. Harvey et al. [42] deployed a randomized algorithm to compute Reeb graph on arbitrary simplicial complexes $K$ in expected time $O(m \log n)$, where $m$ is the size of 2-skeleton of $K$ (i.e., the total number of vertices, edges, and triangles), and $n$ is the number of vertices. For the application of Reeb graphs, Hilaga et al. [43] provide a Multi-resolution Reeb Graph (MRG) representation of triangle meshes which is independent of rotation in topology matching. By reducing the Reeb graph to contour tree via loop surgery, Tierny et al. [76] presented an algorithm to compute Reeb graph on a volumetric mesh in $\mathbb{R}^3$. In a similar vein, the work by Doraiswamy and Natarajan [31] utilizes the union of contour trees to compute the Reeb graph. Other Reeb graph algorithms can be found in [29, 30, 57].

2.4 Mapper

The Mapper is a relatively new concept introduced by Gurjeet Singh, Facundo Mémoli, and Gunnar Carlsson in 2007 [70]. It is an efficient topological method to reduce high dimensional datasets into a graph by applying a topological lens to data that captures specific limited topological information of the high dimensional datasets at a specified resolution.

Mapper construction (see Figure 2.4) starts with a dataset $X$ and a topological lens, real valued function $f : X \rightarrow \mathcal{R}$, on the data. We choose the open covering of the range
of $f$, with overlapping of the neighboring covers. Then, the pre-image of each open set in the covering is an open cover of the dataset $X$. The output graph vertices are created from connected components, found via clustering, within each topological cover (i.e., open set). In other words, the connected components of one open set are “collapsed” into output graph vertices. The output graph edges are added between components that contain the same points from $X$. The resulting graph can describe the overall topology of the connected components of the dataset under the provided topological lens.

![Diagram](image)

Figure 2.4. Mapper on a point cloud model. Using Mapper to construct a topological skeleton, first the (a) point cloud-based model is (b) sliced. (c) Connected components are collapsed to vertices and edges added for components that touch.

The first algorithm for Mapper construction comes in its original paper [70]. [41] provided a provably correct algorithm of parallel computation of Mapper and reported performance experiments on the efficiency of distributed Mapper. Mapper is also one of the main algorithms underlying the tools sold by Ayasdi AI LLC [3], though their implementation is not public.
Mapper can be observed to be an approximation of a Reeb graph. Both the Reeb graph, including the contour tree, and Mapper essentially provide a topological description of the dataset, but their representations of the topological information are different in the following sense. The nodes of the Reeb graph represent the critical points of the underlying scalar field, and the edges of Reeb graph represent the collapsed connected components in the domain where there is no topological change. On the other hand, the nodes in Mapper represent one cluster of data points and the edges are formed because of the common data points in two pre-images from neighboring projection intervals.

2.5 Critical Point Pairing

Persistent homology [32] is the main concept in TDA. Within the framework of TDA, the individual topological features in the Reeb graph, such as cycles, can be ordered according to their birth time and death time (i.e., the function value where the feature appears and disappears). The birth and death times of topological features (equivalently, the paired critical points) can be seen as a signature for Reeb graph [13].

Here we provide a brief and limited introduction to persistent homology, since the necessary concepts from algebraic topology are quite involved. Fortunately, we have an intuitive explanation of topological persistence in contour trees and Reeb graphs in Chapter 4, when we describe an effective algorithm to compute the critical point pairs.

2.5.1 Persistent Homology and Persistence Diagram

The concept of persistent homology was developed by Edelsbrunner et al. [35]. Here we present the theoretical setting for the computation of the persistence diagram associated with a scalar function defined on a triangulated topological space. We then show how this is related to the critical point pairing on Reeb graphs in Chapter 4. We start by presenting a concise description of persistent homology of a finitely triangulable topological space with a
Let $X$ be a triangulable topological space, and let $f : X \to \mathbb{R}$ be a continuous function defined on it. Let $r \in \mathbb{R}$. We denote the sublevel set of $f$ by $X_{\leq r} = \{ x \in X \mid f(x) \leq r \}$. Similarly, we denote the superlevel set of $f$ by $X_{\geq r} = \{ x \in X \mid f(x) \geq r \}$. Let $H_p(X)$ is the $p$-th homology group of $X$. We consider homology with coefficients in a finite field, so $H_p(X)$ is a vector space. Let $\{r_1, \ldots, r_n\}$ be a finite strictly increasing sequence of real numbers. Consider the 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 $X_i = X_{\leq r_i}$ and each homomorphism $g^{i+1}_i : H_p(X_i) \to H_p(X_{i+1})$ on the homology groups is induced by the inclusion $X_i \hookrightarrow X_{i+1}$. We can define $g^j_i : H_p(X_i) \to H_p(X_j)$ for any $i \leq j$ by composition. We say that a class $\alpha$ is born at $r_i$ if:

$$\alpha \in H_p(X_i) \quad \text{but} \quad \alpha \notin \text{im} g^i_{i-1}.$$

A class $\alpha$ born at $r_i$ dies at $r_j$ if:

$$g^j_i(\alpha) \in \text{im} g^j_{i-1} \quad \text{but} \quad g^{j-1}_i(\alpha) \notin \text{im} g^{j-1}_{i-1}.$$

In this case, the function value pair $(r_i, r_j)$ is called a persistence pair, and the difference $r_j - r_i$ is the persistence of the pair. When no confusion occurs, the pairing of function values induces the critical point pairing in Reeb graphs. In other words, when we have a one to one correspondence between critical points and function values, we just represent persistence pairs using the critical points.
These birth and death events are captured by the notion of persistent homology. Specifically, the $p$-th *ordinary persistence diagram* of $f$ is a scatterplot, including multi-set of pairs $(b, d)$, where $b$ and $d$ are corresponding to the birth function value and death function value of some $p$-dimensional homology class, respectively. We denote the $p$-th *ordinary persistence diagram* of $f$ by $D_{g^p}(f)$. In general, the homology group $H_p(X)$ is non-trivial. We call a non-trivial homology class in $H_p(X)$ an *essential homology class*, they will never die during the sequence in Equation 2.1. These non-trivial homology classes are express the cyclic features in Reeb graph. By augmenting an array of relative homology groups to Equation 2.1, we have the following sequence:

$$
0 = H_p(X_0) \to \cdots \to H_p(X_n) = H_p(X) = H_p(X, X_{r_n}) \to H_p(X, X_{r_{n-1}}) \to \cdots \to H_p(X, X_{r_0}) = 0.
$$

(2.2)

Since the final vector space $H_p(X, X_{r_0}) = 0$, every essential homology class eventually dies in some relative homology group $H_p(X, X_{r_j})$. In other words, each essential homology class in a homology group $H_p(X_i)$ will die at some relative homology group $H_p(X, X_{r_j})$, so we have a pair $(r_i, r_j)$, with $r_i \geq r_j$ by the property of relative homology group. We call the scatterplot of multi-set of these pairs the *$p$th extended persistence diagram*. We denote it by $ExD_{g^p}(f)$. In other word, for each point $(b, d)$ in $ExD_{g^p}(f)$, there is an essential homology class in $H_p(X)$, which is born in $H_p(X_{\leq b})$ and dies at $H_p(X, X_{\geq d})$. Observe that for the extended persistence diagram the birth value $b$ is greater than or equal to death value $d$.

To compute these pairs, branch decomposition was first used to provide a multi-scale view of critical points in contour trees [59]. This provides the framework for pairing non-essential critical points in a Reeb graph. The first known description of pairing critical points of a Morse function on a 2-manifold, including essential critical points, is given in [2]. However,
the description is high level with no specific algorithm provided. A similar description of persistence pairing algorithm is also seen in [8].

After the critical points are paired, they can be visualized using a persistence diagram (see Figure 2.5). These scatterplot charts show each pair as a single point, parameterized by birth on the horizontal axis and death on the vertical axis. In addition to serving as a signature for the topology of the input data, a major advantage of persistence diagrams is simplicity and scalability—a large Reeb graph can be reduced to a much easier to interpret scatterplot.

![Persistence Diagram](image)

Figure 2.5. Example of persistence diagram. The persistence diagram, $D_{g_0}(f)$, for the Reeb graph in Figure 2.3 shows topological feature as points, parameterized by birth horizontally and death vertically.

Pairing of critical points of a scalar function has found multiple applications including segmentation of deformable shapes [71], hierarchical shape segmentation [64], description of protein shape [87], automatic extraction of surface structures [84], and 3D shape description and matching [11].
Chapter 3: Parallel Computation of Merge Tree

This chapter\(^1\) introduces the data structure of merge tree and its parallel computation.

3.1 Introduction

Scalar fields are used to describe a variety of details from photographs, to laser scans, to x-ray, CT, or MRI scans. These scalar fields are invaluable for a variety of tasks, such as fatigue detection in machine parts, medical diagnosis, etc. However, analyzing scalar fields can be quite challenging due to their size, complexity, and the need to understand both local details and global context.

The merge tree is the key data structure used in the computation of the join tree, split tree, and contour tree [16]. However, computing these trees is expensive, and their incremental construction makes parallel computation nontrivial.

In its naive implementation, the algorithm to compute merge trees seems efficient. No matter the dimension of the data, it has an $O(n \log n)$ sort phase and an $O(n+k)$ computation phase, where $n$ is the number of elements in the scalar field and $k$ is the aggregate cost of the find operation of a disjoint-set data structure. However, this algorithm has three practical challenges. First, as the dimension of the field is doubled, the number of elements grows quadratically in 2D fields and cubically in 3D fields. Secondly, although asymptotically small, the actual compute time per element in the computation phase is very high. Third, the

\(^1\)Part of this chapter was published in Computer-Aided Design and Applications (2018) [67]. Permission to reproduce in the dissertation is included in Appendix A.
computation phase requires partially ordered incremental construction, making it a challenge to parallelize.

While the global sort can be avoided [62], the algorithm is still difficult to parallelize. Three strategies have been proposed to parallelize merge tree calculations: pruning [18], spatial-domain parallelization [49, 50], and value-domain parallelization [39]. Pruning (see Figure 3.1(a)) works by eliminating elements from the computation, which are predetermined not to be a local minimum, local maximum, or saddle point. This process can be done in parallel, but the compute phase still needs to be completed in serial. Spatial-domain parallelization (see Figure 3.1(b)) divides the scalar field into multiple smaller fields, each distributed to a different thread, processor, or computer. After each sub-field has its tree computed, a messy tree merging process takes over. Finally, value-domain decomposition (see Figure 3.1(c)) distributes the scalar field to different threads, processors, or computers by selecting ranges of element values. This allows parallelizing the loosely ordered computation phase but still requires processing every element.

Each of these approaches takes advantage of certain properties of merge tree construction, but up until now, these strategies have not been effectively integrated. In this chapter [67], we have combined these strategies in an OpenCL merge tree implementation. The implementation results in an $O(n + k)$ pruning phase, an $O(n)$ critical point extraction phase, an $O(c \log c)$ sorting phase, and an $O(c)$ propagation phase, where $n$ is the number of data points, $k$ is the aggregate cost of the disjoint-set data structure, and $c$ is the number of critical points. What’s more, these phases are designed to be parallelized such that they require at worst $O(k)$, $O(1)$, $O(\log c)$, and $O(\log c)$ parallel iterations, respectively. The result is a significant speedup, making computation of trees on large data practical on even modest commodity hardware.
Figure 3.1. Three existing strategies used to parallelize merge tree construction. (a) In pruning, a parallel operation can prune away most non-critical points from computation. (b) In spatial-domain decomposition, regions of the original scalar field are split and distributed to different processes and later reassembled. (c) In value-domain decomposition, elements are distributed to processes based upon ranges of values and later reassembled.

3.2 Conventional Merge Tree Construction

The conventional merge tree construction process [16] starts with a scalar field (see Figure 3.2). The elements of the field are first sorted (see Figure 3.3(a)) in ascending order for a join tree or descending order for a split tree.

![Figure 3.2](image)

Figure 3.2. Example of 2D scalar field used to describe our parallelization of merge tree construction.

The elements are then processed one-by-one. The top element of the list is selected. A tree node is created for that element (see Figure 3.3(b) bottom) and a color assigned (see Figure 3.3(b) top). Next, the neighborhood of eight surrounding elements is searched. If none has been assigned a color (e.g., Figures 3.3(b) and 3.3(d)), the operation is complete. If one (e.g., Figure 3.3(c)) or more (e.g., Figure 3.3(e)) neighbors has already been assigned
A merge tree is formed by removing non-critical point nodes from the tree. This is done by checking each child node in the tree. If the child only has one child of its own (i.e., only one grandchild), then that point is not critical and can be skipped. In Figure 3.3(e) bottom, the node 4 has children 2 and 3. Node 2 has only one child, node 1, while node 3 has zero children. Having a single child means node 2 is not critical, and thus it can be removed. It is removed by connecting node 4 to node 1 (see Figure 3.3(f) bottom).

From an implementation standpoint, this entire operation relies on two algorithmic components. First, the sorting can be handled by any sorting algorithm. Second, the coloring of the nodes is made efficient using the disjoint-set data structure, which has a cost of $O(\alpha(n))$ per lookup, where $\alpha$ is the inverse Ackermann function, an extremely slow growing function.
Other operations are constant time per element. Unfortunately, this strictly-ordered bottom up construction of the tree means that each operation relies upon the results of the prior operation, making parallelization challenging.

3.3 Parallel Merge Tree Construction

Due to the complicated bottom up construction, efficient parallelization requires deconstructing and reordering the operations of the merge tree algorithm. The first two phases of the new implementation are pruning and critical point extraction phases, which uses a spatial-domain decomposition to exclude many of the non-saddle point elements from further computation. In the third phase, the saddles must be sorted. Finally, the critical points are connected by using a value-range decomposition, building subtrees in parallel and propagating their merge information globally. The phases of the algorithm are illustrated in Figure 3.4.

3.3.1 Phase 1: Coloring

The coloring phase has two main objectives. The first objective is to prepare for eliminating as many non-critical points as possible from further computation by using a water shedding approach. The second is to perform a spatial-domain decomposition of the data, by taking the 2D scalar field and splitting it into subfields that can be processed in parallel. The water shedding approach is illustrated in Figure 3.4(a) using the scalar field from Figure 3.2. The first step is to point each element towards its largest (or smallest, depending upon join or split tree) neighbor. If an element is larger than all its neighbors (i.e. a local maxima), it points to itself and is assigned a color. In the next step, each element is updated to the pointer of its pointer. This is essentially the find algorithm of a disjoint-set. This process is repeated until the pointer reaches a colored element, at which point, the element receives that color. Spatial-domain decomposition is accomplished by dividing the scalar field into
Figure 3.4. Illustration of the four phases of parallel merge tree construction. The four phase of our parallel implementation include (a) spatial-domain decomposition and pruning, (b) potential critical point extraction, (c) potential saddle point sorting, and (d) value-domain decomposition of saddle points and propagation of merges.

2D blocks. To complete the processing, neighboring blocks of elements only need to share boundary information. In other words, all elements are computed up to the boundary of their block, all boundaries are synchronized, and then element processing is finalized.

3.3.2 Phase 2: Potential Critical Point Extraction

The merge tree will only contain critical points, so extracting potential critical points early in the process will save computation time. Local minima, maxima, and possible saddle points can be identified by looking at the value of an element relative to its neighbors. Figure 2.1 shows functions which have a local minimum, local maximum, and a saddle point, respectively. A simple observation helps us understand how to detect these three cases.
For the minimum and maximum, notice that all regions surrounding the critical point are higher or lower, respectively. So, if the value of an element is smaller than all its neighbors, it is a local minimum. If the value of an element is larger than all its neighbors, it is a local maximum. The saddle point is a little more complicated to understand. Notice that around the saddle point, the function value goes up in two directions and down in two other directions. Therefore, if the neighbors of an element are larger in two or more disjoint directions and smaller in two or more disjoint directions, then the point may be a saddle. This criterion does not guarantee a saddle point because of interpolation error. However, it can be used to exclude non-saddle points.

Figure 3.4(b) shows four examples. In the first two examples, elements 6 and 4 are each surrounded by two disjoint positive and negative directions. This indicates that these points may be saddles. For element 8, only one disjoint positive and one disjoint negative direction exist. This point can be excluded as non-critical. Finally, for element 3 all neighbors are larger indicating a local minimum.

3.3.3 Phase 3: Saddle Sorting

Merge trees need be built bottom-to-top. At this point in the processing, the extracted coloring information and extracted saddle points (not the minima or maxima) come into play. After the critical points are extracted, the saddle points are colored by looking at the color of all neighboring elements. In Figure 3.4(c), the possible saddle 4, 6, and 7 are extracted. They are colored with their neighbors, with 4 and 6 being colored orange and blue, and 7 being colored only orange. This coloring information identifies which extrema a saddle point potentially connects to. Therefore, 4 and 6 possibly connect the orange and blue extrema, 1 and 3. However, 7 only connects to orange, extrema 1. This means that 7 is not a true saddle point.

Once the coloring is complete, the remaining saddles are sorted by their values.
3.3.4 Phase 4: Subtree Building and Merge Propagation

The final phase of processing builds the tree by performing a value-domain decomposition, which is used to build subtrees and propagate merges. The value-domain decomposition divides the sorted list of critical points into groups, which are each processed in parallel. Building the subtree and propagating merges is a 3-step process. First, the color of nodes is updated with the global recoloring information. Second, subtrees are built using their color information as a guide. Third, the global merge information is updated based upon the new subtrees. This process is repeated until no additional modifications to color occur.

Figure 3.4(d) shows this process. On the left, nodes 4 and 6 are value-domain decomposed into two processing groups with one node each. Each node is updated with the global merge information, which is initially empty (see Figure 3.4(d) top left). The two subtrees are built and the global merge information updated (see Figure 3.4(d) top right). In the second pass (see Figure 3.4(d) right), each group is updated with the new coloring information. For node 4, no changes occur. For node 6, it is only colored purple and is therefore excluded from further computation. At that point, the processing would stop.

3.4 Extension to 3D and Higher Dimensions

Extending this approach to 3D or higher data is mostly trivial. Phases 1 and 2 do require modification. For phase 1 the process is the same, except that now the number of neighbors that must be searched grows to 26 for 3D and much larger for higher dimensions. Phase 2 is problematic since saddle point detection in 3D or higher dimension is complex. This is because there are many more saddle point configurations in higher dimensions. To overcome this, phase 2 saddle detection could be skipped, and all points can be colored and treated as saddle points. The benefit of this is that complex saddle point detection is avoided. The
downside is that a much larger number of saddle points are considered in phases 3 and 4. Finally, phases 3 and 4 can continue unmodified.

### 3.5 OpenCL Implementation

We have implemented the described methods using OpenCL for fast flexible cross-platform interoperability. For phases 1 and 2, each element of the scalar field receives its own thread. The spatial-domain decompositions are square and as large as the supported thread block size of the hardware. For phase 3, each potential saddle point receives its own thread. To sort points in parallel, we used a hybrid of histogram sorting for a rough global ordering and bitonic sorting for precise ordering. For phase 4, each potential saddle point receives its own thread, with the hardware thread block size defining the granularity of the value-domain decomposition.

### 3.6 Experiments

We tested our implementation by comparing it to an optimized C++ implementation of the conventional approach by Carr et al. [16]. We used this conventional implementation to compare the performance and check the correctness of the output tree from our OpenCL approach.

#### 3.6.1 Random Field Tests

To test our approach, we extract the join tree from randomly generated fields. For each of 13 levels of resolution (32 × 32 up to 2048 × 2048), we record the time for 10 different fields (130 tests). A random field represents the most challenging case for calculating merge trees, as it is likely to produce a very dense set of critical points. To test our approach under less dense situations, we analyze those 130 random fields under seven different levels
of smoothing for a total of 910 tests. Random fields have high critical point density, while smoothed fields do not. We report the results from an early 2015 MacBook Pro with an Intel 2.7GHz i5 and an Intel Iris 6100 GPU and a Linux workstation with an Intel 3.4GHz i7 and NVIDIA Tesla K40 Accelerator.

Figure 3.5(a) shows an example 32 × 32 noisy scalar field. This scalar field has 206 critical points, making the tree difficult to display. The scalar field after two and four smoothing iterations can be seen in Figure 3.5(b) and 3.5(c), respectively. These have 98 and 46 critical points, respectively.

![Figure 3.5(a)](image1)
![Figure 3.5(b)](image2)
![Figure 3.5(c)](image3)

Figure 3.5. Example of noisy scalar fields used for performance testing. Example of 32 × 32 scalar fields used to test the performance of our merge tree algorithm: (a) random noise input, (b) after two smoothing iterations, and (c) after four smoothing iterations.

Figures 3.6(a) and 3.6(b) show log-log charts highlighting the performance of various phases of our approach. Figure 3.6(a) shows that in the average case, phases 1 and 2 grow linearly with respect to the number of elements in the field ($R^2 = 0.96$ and $R^2 = 0.974$, respectively). Similarly, Figure 3.6(b) shows that for the average case, phases 3, 4, and OpenCL overhead (data transfer, etc.) grows linearly with respect to the number of critical points in the field ($R^2 = 0.98$, $R^2 = 0.992$, and $R^2 = 0.992$, respectively).

Figures 3.6(c) and 3.6(d) use log-log charts to compare the performance of our approach to the CPU implementation. Figure 3.6(c) shows the computational time against the num-
Figure 3.6. Charts of the performance of our parallel merge tree algorithm on random fields. (a) Log-log chart of the processing time of Phase 1 and 2 is highly linear against the number of elements. (b) Log-log chart of processing time of Phase 3, 4, and overhead is highly linear against the number of critical points. (c) Log-log performance comparison of the number of elements against time and (d) the number of critical points against time. For both hardware configurations, the OpenCL implementation shows on average around 1 order of magnitude improvement over the CPU counterpart. (e) Log-linear boxplots of the speedup based upon the number of element and (f) based upon the number of critical points show that as the problem size grows, the speedup increases as well.
ber of elements, while Figure 3.6(d) shows the computational time against the number of critical points. The average time performance for both our algorithm and the conventional implementation is approximately linear. Our approach has the advantage of being highly parallel in nature. For both hardware configurations, our OpenCL implementation beat the CPU implementation by approximately one order of magnitude.

Figures 3.6(e) and 3.6(f) use log-linear charts to compare the speedup of our approach to the CPU implementation on the MacBook Pro. Figure 3.6(e) shows the speedup against the number of elements, while Figure 3.6(d) shows the speedup against the number of critical points. Interestingly, as the problem size grows, the GPU implementation speedup grows as well. We believe this is caused by the fact that the overall GPU performance is driven by the number of critical points, while the CPU performance is driven by the number of elements.

3.6.2 Contour Trees in Radio Astronomy Data

In radio astronomy, scalar fields are one of the primary data sources used to validate hypotheses. Radio telescopes capture 3D maps of the radio signals in the sky. Two dimensions of these maps are spatial positions in the sky. The third dimension is different radio frequencies. Unfortunately for astronomers, the radio signals collected are very low power and have a low signal to noise ratio. The problem was described best by one radio astronomer, “a cell phone on the moon would be a brightest signal in the sky”.

Figure 3.7(a) shows an example of this data for a single radio frequency. The red blob towards the middle of the image is the feature of interest. In this dataset, this blob represents the signal put off by dust circling a black hole. For our experiments, we calculate the contour tree, which is just the union of a two merge trees (a join and split tree). Figure 3.7(b) shows a small region with the contour tree nodes highlighted. In this image, leaves can be seen (both local minima and maxima) as blueish purple nodes, and saddle points are yellow for join saddles and magenta for split saddles. Figure 3.8 shows the performance results for our
Figure 3.7. Visualization of Radio Astronomy Data. (a) The noisy data shows the amplitude of the radio signal at multiple locations in the sky for a single wavelength. (b) The visualization of the contour tree shows the result of the union of a join tree with a split tree. The density of critical points in this data is quite high.

experiments. These experiments were only run on our MacBook Pro CPU/GPU. For each resolution ($1024 \times 1024$, $2048 \times 2048$, and $4096 \times 4096$), we ran our tree construction on each of $38 - 2D$ slices (radio frequencies) of the data. Considered in our calculations are only the merge tree costs (i.e., no overhead). The log-log chart in Figure 3.8(a) shows that the time taken for our GPU implementation significantly outperforms the CPU implementation. Furthermore, both the CPU and GPU implementations performance grows linearly with the number of elements ($R^2 > 0.99$). The log-linear boxplots in Figure 3.8(b) show the speedup for our implementation. As the number of elements grows, so too does our speedup, reaching on average $40\times$ faster for the GPU implementation on the $4096 \times 4096$ example. The speedup seen here is significantly better than that observed in the random field case. As mentioned in those tests, the random field example is the most challenging because of critical point density. For the random field tests, the median density was one critical point per 33.2 elements. For the radio astronomy data, the median density was one critical point...
per 107.5 elements, over three times less dense. Given the strong relationship between the number of critical points and overall performance of our approach, this result makes sense.

Figure 3.8. Performance on Radio Astronomy Data. Radio astronomy data (a) compute time and (b) speedup on the MacBook Pro CPU and GPU for $38 - 2D$ slices at three different resolutions, $1024 \times 1024$ (1,048,576 elements), $2048 \times 2048$ (4,194,303 elements), and $4096 \times 4096$ (16,777,216 elements).

### 3.7 Conclusions

In conclusion, we have presented an approach for efficiently calculating merge trees in parallel by combining three approaches, pruning, spatial-domain parallelization, and value-domain parallelization. This approach makes it feasible to quickly calculate join trees, split trees, and contour trees for large scalar fields in any number of dimensions. These data structures are incredibly useful for analyzing scalar field data. We have evaluated our approach with a synthetic random field dataset and with a dataset from the discipline of radio astronomy. Although we have calculated the merge tree in parallel, in the future, parallelizing several additional computations would be exceedingly useful. For example, parallelizing the union of merge trees (the last step of building contour trees), calculating of persistence, or the hierarchical simplification of a join, split, and contour trees would all be very useful moving forward.
Chapter 4: Critical Point Pairing in Reeb Graphs

This chapter\textsuperscript{2} introduces efficient algorithms to pair critical points using persistent homology.

4.1 Introduction

The last two decades have witnessed great advances in methods that rely on topological techniques to analyze data using Topological Data Analysis (TDA). The popularity of topology-based techniques is due in large part to their robustness and their applicability to a wide variety of datasets and scientific domains [51]. The \textit{Reeb graph} [63] was originally proposed as a data structure to encode the geometric skeleton of 3D objects, but recently it has been re-purposed as an important tool in TDA.

The Reeb graph encodes the evolution of level sets obtained from a scalar function by sweeping from negative infinity to positive infinity and tracking the birth and death of the connected components of the level sets.

Beside their usefulness in handling large data [34], Reeb graphs and their non-looping variant, contour trees [12], have been successfully used in image processing [45], data simplification [17, 68], feature detection [73], shape understanding [5], visualization of isosurfaces [6] and many other applications. One challenge with using Reeb graphs to directly analyze data is that the graph may still be too large or complex to directly visualize, therefore requiring further abstraction.

\textsuperscript{2}Part of this chapter was published in Advances in Visual Computing (2019) [78]. Permission to reproduce in the dissertation is included in Appendix A.
A fundamental tool in TDA is persistent homology, introduced by Edelsbrunner et al. [35]. Typically, persistent homology operates by transforming a point cloud data into a filtration (a nested sequence of spaces), performing persistent homology computation on the filtration, and parameterizing the obtained topological structures by their life-time in the filtration. As a result, persistent homology gives 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 pair the critical points in the Reeb graph [2].

Figure 4.1 shows an example of this analysis. Initially, a mesh with a scalar function (see Figure 4.1(a)) is converted into a Reeb graph (see Figure 4.1(b)). After that, the critical points are paired, and the persistence diagram displays the data, as seen in Figures 4.1(c) and 4.1(d). This final step can still be challenging, particularly when considering essential critical points—those critical points associated with cycles in the Reeb graph. These require an expensive search that needs to be performed on each essential critical point. While many prior works have provided efficient algorithms for the calculation of Reeb graph structures themselves, to our knowledge, none have provided a detailed description of an algorithm for pairing critical points.

In this chapter, we describe and implement two efficient algorithms to compute persistence diagrams from Reeb graphs. Our first algorithm uses a multi-pass approach that first pairs non-essential critical points using branch decomposition on join and split trees. It then pairs essential critical points using an approach also based upon join trees. Finally, this leads to our second approach, a new single-pass algorithm for pairing both non-essential and essential critical points in Reeb graphs.
Figure 4.1. Reeb graph and persistence diagram. Topological data analysis using Reeb graphs shows (a) data with a scalar function being processed into (b) a Reeb graph. Using the Reeb graph, critical points are then paired. (c) The persistence diagram, $D_{g_0}(f)$, and (d) extended persistence diagram, $ExD_{g_1}(f)$, provide a visualization of the structures in the original data.

4.2 Reeb Graph

In Chapter 2 we introduced persistent homology and the persistence diagram. We also discussed critical point pairing of contour tree in an intuitive manner. Now we generalize the critical point pairing of the contour tree to the Reeb graph and provide a strict mathematical treatment of pairing in persistent homology.
We follow the terminology in Chapter 2. Let $X$ be a triangulable topological space, and let $f : X \to \mathbb{R}$ be a continuous function defined on it. We define an equivalence class by the relation $\sim$ on $X$, such that $x \sim y$, if and only if $x$ and $y$ belongs to the same connected component of $f^{-1}(r)$ for some $r \in \mathbb{R}$. Given $X$ and a function $f : X \to \mathbb{R}$, the Reeb graph of $X$ and $f$ is the quotient space $X/\sim$ equipped with the quotient topology induced by the quotient map $\pi : X \to R_f(X)$. The Reeb graph is denoted by $R_f(X)$. When $X$ is clear from the context, we will denote the Reeb graph simply by $R_f$.

The Reeb graph 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. Figure 4.1 (a) and (b) shows an example of a Reeb graph defined on a surface.

The function $\tilde{f}$ 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 incident to $x$ which have greater values of $\tilde{f}$ than $x$. The down-degree is defined similarly. A point $x$ on $R_f$ is a critical point if either its up-degree or down-degree is not equal to one. Otherwise it is a regular point. A critical point on the Reeb graph is also a node of Reeb graph. A critical point is a minimum or maximum if its down-degree or up-degree is equal to 0. Finally, a critical point is called a down-fork or up-fork if its down-degree or up-degree is greater than 1.

Without loss of generality, we assume that Reeb graph is a single connected component, and each node on Reeb graph has unique function value. Moreover, we assume also that every node in the Reeb graph is either a up-fork with up-degree 2, an down-fork with down-degree 2, a maximum, or a minimum. This is not a restriction to the general case, since a Reeb graph that does not satisfy these requirements can be conditioned to fit them, as we will show in Section 4.4.

We also assume that the scalar function $f$ is tame in the sense of [8].
4.2.1 Persistence Diagram of Reeb Graph

Of particular interest to us are the persistence diagram $D_{g_0}(f)$ and extended persistence diagram $\text{Ex}D_{g_1}(f)$. These two diagrams can be computed completely by considering the Reeb graph $R_f$. We give an intuitive explanation to this fact here, and we refer the reader to [8] for more details.

Note that pairing of critical points of a scalar function can be computed independent of the computation of Reeb graphs. However, the pairing is best described using Reeb graph since the structure of Reeb graph clearly reveals the topological feature associated to the pairing.

Before we describe the points in the persistence diagram, $D_{g_0}(f)$, and extended persistence diagram, $\text{Ex}D_{g_1}(f)$, we need to distinguish between two types of forks in the Reeb graph, namely the ordinary forks and the essential forks. Let $R_f$ be a Reeb graph and let $o$ be a down-fork such that $r = f(o)$. We say that the down-fork $o$ is an ordinary fork if the lower branches of $o$ are not from the same connected component of $(R_f)_{<r}$. The down-fork $o$ is said to be essential if it is not ordinary. The ordinary and essential up-forks are defined similarly.

4.2.1.1 Branching Features of Reeb Graph

We first consider pairing ordinary down-forks using sublevel set filtration. Let $r \in \mathbb{R}$. We track changes that occur in $H_0((R_f)_{\leq r})$ as $r$ increases. A connected component of $(R_f)_{\leq r}$ is created when $r$ passes through a minimum of $R_f$. Let $C$ be a connected component of $(R_f)_{\leq r}$. We say that a local minimum $m$ of $R_f$ creates $C$ if $m$ is the global minimum of $C$. Every ordinary down-fork is paired with a local minimum to form one point in the persistence diagram $D_{g_0}(f)$ as follows. Let $o$ be an ordinary down-fork with $f(o) = r$ and let $C_1$ and $C_2$ be the connected components of $(R_f)_{<r}$. Let $c_1$ and $c_2$ be the creators of $C_1$ and $C_2$, and
assume that \( f(c_1) < f(c_2) \). The homology class \([c_1 + c_2]\) that is born at \( f(c_2) \) and dies at \( f(o) = r \) generates a point \((c_2, o)\) in the 0-th ordinary persistence diagram \( Dg_0(f) \). Note that such a pair occurs when the minimum is a branch in the Reeb graph, hence we name it a branching feature. Note also that we use critical points to represent the persistence pair, not the function values here.

Ordinary up-forks can be paired similarly by using superlevel set filtration. The pairing of each up-fork with a local maximum gives rise to points in the 0-th persistence diagram \( Dg_0(f) \). For an ordinary up-fork, \( u \), with \( f(u) = r \), connected components \( C_1 \) and \( C_2 \) now come from \((R_f)_r^>\). Let \( c_1 \) and \( c_2 \) be the creators of \( C_1 \) and \( C_2 \), and assume that \( f(c_1) < f(c_2) \), the homology class \([c_1 + c_2]\) that is born at \( f(c_1) \) dies at \( f(u) = r \) and generates a point \((c_1, u)\) in \( Dg_0(f) \).

4.2.1.2 Cycle Features of Reeb Graph

Let \( s \) be an essential down-fork with \( f(s) = r \). We say \( s \) a creator of a 1-cycle in the sublevel set \((R_f)_r^\leq\). As shown in [2], \( s \) will be paired with an essential up-fork \( s' \) to form an essential pair \((s', s)\), a point in the extended persistence diagram \( ExDg_1(f) \). The essential up-fork \( s' \) is determined as follows. Let \( \Gamma_s \) be the set of all cycles born at \( s \) in the Reeb graph \( R_f \). We can pick the largest one among all the minimums of each cycle in \( \Gamma_s \). Then \( s' \) is the point at which the function \( f \) achieves this largest minimum [8].

4.3 Related Work

Pairing of critical points of a scalar function has found multiple applications including segmentation of deformable shapes [71], hierarchical shape segmentation [64], description of protein shape [87], automatic extraction of surface structures [84], and 3D shape description and matching [11].
Branch decomposition was first used to provide a multiscale view of contour trees [59]. This provides the framework for pairing non-essential critical points in a Reeb graph. The first known description to pair critical points of a Morse function on a 2-manifold, including essential critical points, is given in [2]. However, the description is high level with no specific algorithm provided. Similar description of persistence pairing algorithm is also seen in [8].

To the best of our knowledge, this is the first systematic development and implementation of two intuitive and efficient algorithms to pair the nodes of Reeb graphs by persistent features.

4.4 Conditioning the Graph

As mentioned in Section 4.2.1, 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.

![Figure 4.2](image-url)

(a) Non-Critical  (b) Degenerate Maximum  (c) Double Fork  (d) Complex Fork

Figure 4.2. Before pairing, the nodes of Reeb graph must be properly conditioned. There are four node configurations that require conditioning. New nodes and edges are shown in blue.

There are four 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 Figure 4.2(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 Figure 4.2(b), this condition is corrected by added a new node for the local maximum $\epsilon$ higher value, where $\epsilon$ 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 [70].

• 2:2 nodes—Nodes with both 2 up- and 2 down-degree are degenerate double forks, both down-fork and up-fork. Figure 4.2(c) shows how double forks can be corrected by splitting into 2 separate forks, one up- and one down-fork, $\epsilon$ distance apart.

• 1:N $> 2$ (and N $> 2$:1) nodes—Nodes with down-degree (or up-degree) 3 or higher, are complex forks to pair. These are the forks corresponds 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 one, requiring complicated tracking of pairs. To simplify this, as seen in Figure 4.2(d), complex forks can be split into two forks $\epsilon$ apart. The upper down-fork retains one 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.

Beyond these requirements, we assume the Reeb graph is a single connected component. If the Reeb graph contains multiple connected components, each one can simply be extracted and processed individually.
Figure 4.3. Example of multipass critical point pairing. In the multipass approach, (a) the Reeb graph has (b) a split tree and (c) a join tree extracted for non-essential pairing. Then in a separate process, the (d) essential forks are paired one at a time. The persistence diagram for this Reeb graph is shown in Figure 4.1(c) and 4.1(d).

4.5 Multipass Approach

Roughly speaking the Reeb graph gives rise to two types of topological features: the branching features and cycle features. These features are precisely encoded in the zero persistence diagram $D_{g0}(f)$ and first extended persistence diagram $ExD_{g1}(f)$ [8]. The persistence diagram $D_{g0}(f)$ can be obtained by pairing the non-essential fork nodes of the Reeb graph. On the other hand, the extended persistence diagram $ExD_{g1}(f)$ can be obtained by pairing of essential fork nodes. We next demonstrate these two steps using Figure 4.3(a) as a running example.
4.5.1 Non-Essential Fork Pairing

Identifying the non-essential forks can be reduced to calculating merge trees, both a join and a split tree, on the Reeb graph (see Figures 4.3(b) and 4.3(c)). In our implementation, this is done using Carr’s et al.’s approach [16].

Next, a stack-based algorithm, based upon branch decomposition [59], is executed to pair critical points. The algorithm operates as a depth first search that seeks out simply connected forks (i.e., forks connected to two 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, one of three operation types occurs, as seen in Figure 4.4. 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 two variants that occur when two 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 (see Section 4.2.1.1). The stack is processed until only a single leaf node remains on it. This will be the global minimum/maximum for the join tree and split tree, respectively.

The algorithm operates identically on both join and split trees. Finally, the unpaired global minimum and maximum left on the join and split tree stacks can be paired.

Figure 4.5 shows an example for the join tree in Figure 4.3(c). 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 two 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 $D_{g0}(f)$ in Figure 4.1(c), along with the split tree pairing, $O/N$, and the global min/max pairing, $A/P$.

4.5.2 Essential Fork Pairing

The remaining unpaired forks are essential forks, as seen in Figure 4.3(d). We extract an algorithms from the high-level description of [8] to pair them. The procedure processes up-forks one at a time.

The essential fork pairing algorithm can be treated as join tree problem. 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.

Figure 4.6 shows the sweeping process for the up-fork $D$. Initially (see Figure 4.6(a)), $D$ is split into $D_L$ and $D_R$, which are each part of separate connected components, denoted by color (see Figure 4.6(b)). As the join tree is swept past $E$ (see Figure 4.6(c)), a new connected component is formed. In Figure 4.6(d), $F$ is added to the connected component of $D_R$. As the join tree is swept past $G$ (see Figure 4.6(e)), the $E$ and $D_L$ connected components join. The process continues until Figure 4.6(h), where three connected components exist. The purple and yellow components join at $K$ (see Figure 4.6(i)). Finally at $L$ (see Figure 4.6(j)), both $D_L$ and $D_R$ are part of the same connected component. This indicates that $D$ pairs with $L$.

Figure 4.7 shows the 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 ExDg$_1(f)$ in Figure 4.1(d).
Figure 4.6. 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.

4.6 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.
Figure 4.7. Essential fork pairing in the multipass algorithm for the example Reeb graph from Figure 4.3. Each up-fork (D, F, and I, respectively) is split into two pieces and (d-f) a join tree calculated from the (a-c) superlevel set to find the partner. Figure 4.6 shows a detailed calculation for D.

4.6.1 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.

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

- For local minima, a label for the current critical point is propagated upward. In the examples of Figure 4.8(a) and 4.8(b), both A and B are propagated to C.
- For local maxima, nothing needs to propagate.
- For down-forks, all unpaired labels are propagated upwards. In the example of Figure 4.8(c), the critical points B and C are paired, thus only A is propagated to D.
- For up-forks, 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 Figure 4.8(d), the labels $A$ and $D_L$ are propagated to $G$, and labels $A$ and $D_R$ are propagated to $F$.

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 local maxima, the list labels is searched for the unpaired up-fork with the largest value. Those critical points are then paired. In the example from Figure 4.9(c) for local maximum $O$, the list is searched and $N_L$ is determined to be the closest unpaired up-fork.

- For down-forks, 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 is closes a cycle with the associated up-fork. In the example, Figure 4.9(a), 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 Figure 4.8(c), no essential up-forks are found for $C$, and the largest local minimum, $B$ is selected instead.

4.6.2 Virtual Edges for Propagate and Pair

The basic propagate and pair approach succeeds in most case, but in certain cases, such as in Figure 4.10(a), it fails. 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
Figure 4.8. Propagate and Pair algorithm on the example Reeb graph from Figure 4.3. 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. The example is continued in Figure 4.9.
Figure 4.9. Continuation of Figure 4.8.

(a) Essential down-fork
(b) Non-essential up-fork
(c) Local maximum
(d) Global min/max

Figure 4.10. An example case of where the basic propagate and pair algorithm fails. In this case (a), $B$ and $F$ should pair but will 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.
Figure 4.11. An example requiring virtual edge merging. (b-c) Virtual edges are created. (d-e) Virtual edges are propagated. (f) At the down-fork \( N \), virtual edges \( V_J \) and \( V_K \) are propagated and merged into \( V_{JK} \).

resolve this communication issue, virtual edges are used. Virtual edges have four associated operations.

Virtual edges are created on all up-fork operations. For example in Figure 4.10(b), when processing \( B \), the endpoints of the fork, \( E \) and \( F \) are connected with virtual edge \( V_B \). Similarly, in Figure 4.10(c), when processing up-fork \( D \), another virtual edge \( V_D \) is created connecting the endpoint, \( E \) and \( F \).

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 Figure 4.10(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 \( D \).
When processing down-forks, all incoming virtual edges need to be pairwise merged. Figure 4.11 shows an example. In particular, in Figure 4.11(f) when processing down-fork $N$, the 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 $J$.

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 Figure 4.10(e). During the virtual edge propagation phase, redundant virtual edges can also be culled. For example, the virtual edge $V_D$ is a superlevel set of $V_B$. Therefore, $V_B$ can be discarded.

The necessity of the virtual edge process can also be seen in Figure 4.8. In Figures 4.8(i)-4.8(l), the pairing of $L$ with $D$ is only possible because of the virtual edge created by $I$ in Figure 4.8(i).

### 4.7 Evaluation

We have implemented the described algorithms using Java. Reported performance was calculated on a 2017 MacBook Pro, 3.1 Ghz i5 CPU, 8 GB RAM. We investigate the runtime performance of the algorithms using the following:

- The Reeb graph in Figure 4.1 was built by hand to demonstrate the functionality of our approach.

- Synthetically generated, split trees, join trees, and Reeb graphs in Figure 4.12. The synthetic examples were generated by a Python script. Given a positive integer $n$, where $n = \{100, 500, 1000, 3000, 5000\}$, the script starts by creating a fork $G_1$ consisting of a node with valency 3 and three nodes with valency 1 linked to the 3-valence node. For each iteration $i < n$, another fork is generated, and one or two of its 1 valency
nodes are glued to the nodes in $G_{i-1}$ with valency 1. If we constrain the choice of gluing a single node at each iteration the resulting graph will be a split tree.

- Reeb graphs calculated on publicly available meshes in Figures 4.13 and 4.14 are 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 in Figure 4.16. The 2016 SciVis Contest data\(^3\) is a large time-varying multi-run particle simulation. Our evaluation took one realization, smoothing length 0.44, run 50, and calculated the Mapper graph for all 120 time-steps using the variable of interest, concentration. The Mapper graph was generated using a Python script that follows the standard Mapper algorithm [70].

The performance for the algorithms can be seen in Table 4.1. 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 explain.

We compared the exact same tree structures as split trees and join trees by negating the function value of the input tree. The resulting persistence diagrams can be seen in Figures 4.12(a) and 4.12(b). The performance observed in Table 4.1 and Figure 4.15(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 has many virtual edges created and processed, while the join tree has none. The extra virtual

\(^3\)https://www.uni-kl.de/sciviscontest/
Figure 4.12. Persistence diagrams for random trees and random graphs. The number indicates how many random iterations were used to generate the example. The up-fork features of $D_{g_0}(f)$ are shown in blue; the down-fork features of $D_{g_0}(f)$ are shown red; and the cycles in $ExD_{g_1}(f)$ are colored purple.

edges are the sole cause of the reduced performance. In fact, split trees are representative of a potential worst case, generating many unused 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.
Figure 4.13. Meshes and Reeb graphs used in evaluation. Meshes (column 1), Reeb graphs (column 2), $Dg_0$ (column 3), and $ExDg_1$ (column 4) for datasets used in evaluation. The mesh is colored by the scalar function applied to it. The $Dg_0$ shows up-forks in blue and down-forks in red. The $ExDg_1$ shows cycles in purple.
Figure 4.14. Continuation of Figure 4.13.

We next investigate the performance of randomly generated Reeb graphs, shown in Table 4.1 and Figure 4.15(b). These Reeb graphs consist predominantly of cycles, as seen in Figures 4.12(c) and 4.12(d). This represents another type of worst case, since the many...
Table 4.1. Performance for all datasets tested. Bold indicates the faster algorithm.

<table>
<thead>
<tr>
<th>Data</th>
<th>Figure</th>
<th>Mesh</th>
<th>Reeb Graph Nodes</th>
<th>Cycles</th>
<th>Multipass Time (ms)</th>
<th>Single-pass Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>random_tree_100</td>
<td>4.13(c)</td>
<td>10401</td>
<td>20814</td>
<td>23</td>
<td>4</td>
<td>2.06e-03</td>
</tr>
<tr>
<td>random_graph_100</td>
<td>4.13(d)</td>
<td>10098</td>
<td>26216</td>
<td>33</td>
<td>6</td>
<td>1.61e-03</td>
</tr>
<tr>
<td>random_tree_500</td>
<td>4.14(b)</td>
<td>24138</td>
<td>62284</td>
<td>8</td>
<td>3</td>
<td>7.82e-04</td>
</tr>
<tr>
<td>random_graph_500</td>
<td>4.14(d)</td>
<td>3070</td>
<td>6144</td>
<td>13</td>
<td>2</td>
<td>5.29e-04</td>
</tr>
<tr>
<td>random_tree_1000</td>
<td>4.14(a)</td>
<td>8410</td>
<td>16816</td>
<td>15</td>
<td>0</td>
<td>7.82e-04</td>
</tr>
<tr>
<td>random_graph_1000</td>
<td>4.14(c)</td>
<td>39994</td>
<td>80000</td>
<td>23</td>
<td>4</td>
<td>8.03e-04</td>
</tr>
</tbody>
</table>

Figure 4.15. Performance on random split/join trees and graphs. Plots of the compute time for various input sizes to (a) the random split/join tree and (b) the random graph for Table 4.1.

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 Figure 4.17, that randomly
cuts $n$ cycles in the starting Reeb graph random_graph_5000 containing 2400 cycles. The result shows that the break even is about 900 cycles (about 25% essential features and 75% non-essential features).

The SciVis contest data was “cycle heavy” as can be seen in the persistence diagram of Figure 4.16. Given the analysis of the random graph, it is unsurprising that the performance of the single-pass approach was slightly lower than that of the multipass approach.

### 4.8 Conclusions

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 a significant advantage in simplicity of implementation. Once a merge tree algorithm is obtained and branch decomposition implemented, the only necessity is repeated calls to that algorithms. This approach also has a potential advantage
in parallelism. First, the join and split trees could be processed in parallel. Then, all essential up-forks could be processed in parallel.

The single-pass algorithm showed a slight edge in performance, particularly for data with a balance between the number of essential and non-essential features. The other significant advantage of the single-pass approach is that it is in fact a single-pass approach, only visiting critical points once. This type of approach 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.
Chapter 5: Contrast and Brightness Enhancement of Color Image

This chapter\textsuperscript{4} presents an application of the contour tree and persistence diagram in image enhancement.

5.1 Introduction

Image enhancement techniques aim to provide maximal control in improving the appearance of an image. They are widely used in photography post-processing (i.e., retouching), medical image processing, and object recognition. Along with the proliferation of image-content producing smartphones, the demand of new techniques to enhance the content continues to grow as well. Image editing techniques mainly utilize region selection, similarity, and/or thresholding to determine groups of pixels to editing, which is adequate in many cases. However, those do not directly consider the topological structure of an image, which can be utilized to capture regions of monotonic behavior (i.e., similar gradient direction), in a hierarchical manner.

In recent years, TDA has been widely used in shape-based analysis of complex data. The main tool of TDA that we use is the contour tree, which was first introduced by Boyell and Ruston [12], who named it the “enclosure tree” of contour lines for the height of terrain. The contour tree of a scalar function defined on a simply connected domain (i.e., an image) is obtained by encoding the evolution of the connectivity of the level sets induced by a scalar function defined on the domain.

\textsuperscript{4}Part of this chapter was published in Advances in Visual Computing (2019) [80]. Permission to reproduce in the dissertation is included in Appendix A.
There are two key properties of contour tree that makes it a viable tool for image segmentation and enhancement. First, contour tree has a graph-based representation that captures the changes of topology in image data. Second, the contour tree can be searched, modified, and pruned in a quantifiable way in order to segment, remove noise, or enhance contrast, while retaining the global structures of image.

In this chapter, we leverage the contour tree to extract a hierarchical representation of key features of an image (i.e., critical points) and the monotonic regions connecting those features. The data provided by the contour tree provides new segmentation functionalities, and with the use of topology-aware transfer functions, it provides new image enhancement functionalities that use local topological properties to operate on pixels, instead of using global image properties.

We first introduce the definition of contour tree on a simply connected domain $\Omega$ of $\mathbb{R}^2$. Let $f : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$ be a continuous function on $\Omega$. The level set of a single isovalue $z$ is the set $f^{-1}(z) = \{(x, y) : f(x, y) = z\}$, and a contour is a connected component of a level set. The most familiar context of contours are topographic maps (see Figures 5.1(a)
Figure 5.2. Contour tree and its decomposition. (a) A low resolution version of the scalar field from Figure 5.1(a) has its (d) augmented join tree and (e) augmented split tree generated. (f) The trees are combined to create the augmented contour tree. The critical points of the augmented trees are used to generate the (b) join tree, (c) split tree, and (see Figure 5.1(d)) contour tree of the scalar field.

and 5.1(c)) [12], where the function \( f \) is the land elevation, and isovalues are shown at selected discrete values.

The contour tree tracks the creation, merging, splitting, and destruction of contours as a plane is continuously swept across \( f \). Consider the example in Figure 5.1— in particular, the height map in Figure 5.1(c) and contour tree in Figure 5.1(d).

First, a plane \( z \) is swept from \(-\infty \to +\infty\). As the plane sweeps up, when it reaches local minima, nodes are created in the contour tree, denoted by labels \( m \), \( l \), and \( d \), since these represent the “birth” of a contour. As the plane continues its sweep up, one can observe that at \( z = f_{red\text{-plane}} \) there are three independent contours, each represented by an edge in the contour tree.

At \( z = f_h \), a special event occurs, where the contour of \( l \) and \( d \) merge together. The merge, called a join event, represents the “death” of the contour that was born more recently,
in this case \( l \). The event creates the feature pair \( l/h \). Similarly, at \( z = f_o \), the contours of \( m \) and \( d \) join together and \( m \) “dies”, creating the \( m/o \) feature pair.

The birth/death relationships are important, because they segment the space into a hierarchy of regions of uniform (i.e., monotonic) behavior. Furthermore, the difference between the birth and death, \( |f_{\text{birth}} - f_{\text{death}}| \), of a contour is known as the persistence of the feature. Persistence is an important measure in our context, as it captures the amplitude/scale of a feature.

Likewise, we also consider a sweep plane \( z \) that goes from \( +\infty \to -\infty \). As the plane sweeps downwards, new contours are born at local maxima, such as \( a \) and \( j \). For downward sweeps, when the contours merge together at \( z = f_f \), this is called a split event. Similarly to join events, the split represents the “death” of the feature born more recently, in this case \( j/f \). It is important to recognize that with splits, \( f_{\text{birth}} > f_{\text{death}} \).

Finally, the global minimum and maximum are paired into a special feature, which represents the range of values. This feature is captured twice as \( d/a \) and \( a/d \).

5.1.1 Computing the Contour Tree

We briefly describe the computation of the contour tree. For a more detailed description and efficient algorithms, see [16, 67].

As the prior description of the contour tree implies, the construction is split into 2 phases: join tree construction (see Figure 5.2(b)) and split tree (see Figure 5.2(c)) construction. The join tree represents the upward sweep, while the split tree represents the downward sweep. To find the join and split trees, the construction first finds the augmented join tree and augmented split tree of the image.

Using the scalar field in Figure 5.2(a), which is a downsampling version of Figure 5.1(a), we will describe the augmented join tree construction in Figure 5.2(d). First, the pixels of the image are sorted by values, \( f_d < f_c < f_m < f_l < ... < f_f < f_j < f_a \). Pixels are
then inserted one at a time into the augmented join tree. As they are inserted, connected components are tracked by connecting with neighboring pixels already in the augmented join tree. In our implementation we consider the ring of eight neighbors surrounding a given pixel. In this illustrative example, we only consider four (i.e., left/right/up/down) neighbors. If a pixel joins one or more existing connected components, it is connected to the top of those components in the augmented join tree. If it joins no connected component, it starts a new connected component. For example, when $d$ is initially added to the tree, no connected components exist, so it creates one. When $c$ is inserted, it joins the \{d\} connected component, since they are neighbors in the image. Continuing forward, when $h$ is inserted, it links the \{l\} connected component to the \{c, d\} connected component.

Augmented split tree construction is performed identically, except that it starts with the largest valued pixel first. The augmented contour tree is constructed by pealing leaf nodes off of the join/split trees and adding them to the augmented contour tree, as described in [16]. The join/split/contour trees consist of only critical nodes (i.e., nodes that cause birth and death events). They are calculated by removing any “regular” node, those having only one downward edge and one upward edge, from the augmented version. Finally, using the contour tree, the critical points are paired using the approach described in Chapter 4 [79].

The construction of contour tree assumes that all pixels have unique values, which is not the case in real images. Equal valued pixels are problematic when they are next to one another because insertion order can change the contour tree structure. This is resolved by grouping equal-valued neighboring pixels into “super-pixel” units that are processed together.
5.2 Capturing the Topology of an Image

5.2.1 Feature Subtree Extraction

Extracting the regions of monotonic behavior requires selecting a subtree from the augmented contour tree for a given feature pair. Given a join/split node and local minimum/maximum pair, starting at the join/split, find the three subtrees extending from it. The selected subtree for the feature pair is the one containing the local minimum/maximum node. For example, in Figure 5.2(f), the \(m/o\) feature pair contains join node \(o\) and local minimum \(m\). Join node \(o\) has three subtrees: up, down-left, and down-right. The down-left subtree contains the local minimum node \(m\), making it the feature subtree, containing nodes \(\{m, i, n, o\}\).

5.2.2 The Contour Tree of Color Images

We now consider mapping a color image to the contour tree. The contour tree requires \(f : X \rightarrow \mathbb{R}\) (i.e., a single color channel). However, considering color images in RGB (Red, Green, Blue) colorspace, three channels map to each pixel. We consider each channel, red, green, and blue independently, generating three contour trees. Our implementation also consider HSB (Hue, Saturation, Brightness) colorspace. Saturation and brightness each map to their own contour trees. However, our approach is unable to properly use hue, as it maps to \(S^1\) (i.e., a circular coordinate system), and the contour tree cannot be applied to such a domain.

5.3 Image Processing via Contour Trees

The basic procedure for topology-based segmentation and enhancement of image is:

- Section 5.3.1: First the process entails selecting a set of feature pairs of interest.
• Section 5.3.2: Next, those feature pairs can be used to segment the image.

• Section 5.3.3: Alternatively, the feature pairs can be used to edit the image. After the edit is applied to the image, the contour trees are recalculate and the process can restart.

5.3.1 Visualizing the Contour Tree

To select feature pairs, each contour tree is displayed using two interactive interfaces—the persistence diagram and persistence-volume diagram.

Direct visualization of the contour tree is generally not advisable, as the size and complexity of the tree become unmanageable for even moderately sized data. The standard practice in TDA is to represent the contour tree using the birth/death feature pairs in a scatterplot display, called a persistence diagram [23]. In a persistence diagram, the x-axis is linked to the feature birth value, while the death value is linked to the y-axis. Figure 5.3(a) shows an example for the contour tree in Figure 5.1(d). Join feature pairs (i.e., \(l/h\) and \(m/o\)) are on the upper left, while split feature pairs (i.e., \(j/f\)) are on the lower right. Feature pairs are also colored by their type—join features blue and split features red. One additional useful property of the persistence diagram is that the distance of the point from the diagonal is an analog of the persistence of the feature pair. This effectively means that features with larger amplitude are farther from the diagonal.

The persistence diagram primarily captures the amplitude of a selected feature. The volume, the size in pixels of a feature, is sometimes important as well. The persistence-volume diagram is an alternative scatterplot representation that encodes persistence (i.e., \(|f_{\text{birth}} - f_{\text{death}}|\)) on the x-axis using a linear scale and volume on the y-axis using a log scale. The volume is calculated by counting the number of nodes/pixels in the feature subtree.
Figure 5.3. Interfaces used for selecting topological features. The main interfaces used for selecting topological features use the (a) persistence diagram and (b) persistence-volume diagram. Once features selected (c) the image editing interface enables selecting the type and scale of the edit function.

For example, the \( m/o \) contains four nodes/pixels, \( m, i, n, \) and \( o \). Figure 5.3(b) shows an example.

Once the contour trees are generated and visualized, user interaction can proceed. We provide a brushing mechanism on both the persistence diagram and persistence-volume diagram for selecting a set of features of interest. As the mouse is clicked-and-dragged, the features and feature subtrees are gathered for further processing. For example, brushing across the middle of the persistence-volume diagram in Figure 5.3(b) would select feature \( m/o \) and its subtree, as shown in Figure 5.4(a). Features of the contour tree are hierarchical, thus, if more than one feature is selected, those features may be inclusions (i.e., one feature may be a subsets of another). In that case, only the larger/outermost feature is processed.
5.3.2 Subtree Selection as Image Segmentation

The selected feature pairs and their associated subtrees are relatively easy to use for segmentation. First, a mask is created at the full image resolution. Next, for each feature subtree, the nodes/pixels contained in the subtree can be marked in the mask. Finally, the mask can be used for a binary segmentation of the image. Figure 5.4 shows how this would work given the selection of the \( m/o \) feature from Figure 5.1. Because this is a binary mask, multiple masks can be generated from different feature selections and boolean operators (i.e., and, or, not, etc.) applied to generate more dynamic segmentations.
5.3.3 Subtree Modification as Image Editing

Once a segmentation mask is built, global image editing options (i.e., contrast enhancement, brightness enhancements, etc.) could be easily executed. However, this misses an opportunity that the contour tree provides for modifying pixels based upon their local topology, by using subtree information for the modification. For subtree-based modification we provide four transfer function options that directly modify the pixels of the subtree, based upon properties of the subtree.

- **Contrast Enhancement:** Contrast enhancement fixes value of the feature pair join/split node and linearly stretches rest of the subtree. For a given node $\sim$ in the subtree and a contrast scale factor $s \geq 1$, the value $f'_{\sim} = f_{\text{death}} + (f_{\sim} - f_{\text{death}}) \times s$. Figure 5.5(a) shows an example of the operation, where the local contrast enhancement fixes the death value of a feature while lowering the birth value.

- **Denoising:** Denoising linearly collapses the subtree, such that all pixels eventually have the same value, $f_{\text{death}}$. The calculation of denoising is directly related contrast enhancement—for a given node $\sim$ in the subtree and a denoising scale factor $0 \leq s \leq 1$, the value $f'_{\sim} = f_{\text{death}} + (f_{\sim} - f_{\text{death}}) \times s$. Figure 5.5(b) shows an example, where once again the death value of a feature is fixed and the other nodes/pixels are increased.

- **Brightness Enhancement:** The brightness of the entire subtree can be modified up or down uniformly. For a given node $\sim$ in the subtree and a brightness scale factor $-255 \leq s \leq +255$, the value $f'_{\sim} = f_{\sim} + s$. Figure 5.5(c) shows an example where all nodes are modified identically.

- **Gamma Correction:** Gamma correction provides a nonlinear correction, which is usually applied to the luminance of an image. For a given node $\sim$ in the subtree and a gamma correction value $\gamma$, where $\gamma > 0$, the value of a pixel $f_{\sim}$ is first normalized and...
Figure 5.5. Four functions based on subtree editing. Based upon the subtree selected in Figure 5.4(c), the operation of (a) contrast enhancement, (b) denoising, (c) brightness enhancement, and (d) gamma correction are shown.
gamma corrected $\overline{f} = \left(\frac{f - f_{\text{death}}}{f_{\text{birth}} - f_{\text{death}}}\right)^\gamma$. The final value of the pixel is then linearly interpolated, $f' = f_{\text{death}} \ast (1 - \overline{f}) + f_{\text{birth}} \ast (\overline{f})$. Figure 5.5(d) shows a gamma correction, where $\gamma < 1$.

The interface for selecting the editing mode and scale is shown in Figure 5.3(c). The interface is designed to represent how the chosen function modifies the persistence of the feature (horizontally) and the birth and death of a feature (vertically). For example, contrast enhancement increases persistence, while denoising decreases it. Brightness enhancement modified both the birth and death of a feature, leaving persistence unchanged. Finally, gamma correction changes nothing about the birth, death, or persistence. The level of the chosen transfer function is selected with a simple slider.

It is important to note that, with the exception of gamma correction, these transformations are not local topology preserving. As part of the normal process of image editing, the contour trees are recalculated after an edit is applied to the image.

5.4 Examples

We have implemented a prototypes of our approach using Java. All images are generated using a 2017 MacBook Pro. All aspects of our approach are interactive, except the construction of the contour trees and extraction of feature pair sub-trees. For the examples shown, contour tree construction and sub-tree extraction took between 5 and 30 seconds, depending upon the size and complexity of the image.

- Synthetic Example: We first examine the edits to our synthetic example, seen in Figure 5.6(a). In Figure 5.4(b), we first observe the segmentation of the m/o feature from Figure 5.4. Based upon that segmentation, the operations outlined in Figure 5.5 are applied, contrast enhancement (see Figure 5.6(b)), denoising (see Figure 5.6(c)), brightness enhancement (see Figure 5.6(d)), and gamma correction (see Figure 5.6(e)).
Figure 5.6. Illustration of four functions on scalar field. (a) The scalar field from Figure 5.1(a) has the four operations of Figure 5.5 applied.

- Grayscale Images: Since contour trees operate on a single channel, grayscale images are a natural way to demonstrate this functionality. For the following datasets, the brightness channel of the HSB colorspace was used for editing.

  - The *Florala dataset*, seen in Figure 5.7(a), is a black and white photograph of Florala Alabama. The data was retrieved from [44]. The photograph had a series of denoising, contrast enhancement, and gamma correction steps applied to recover the final image in Figure 5.7(b). The persistence and persistence-volume diagrams are shown both before and after for comparison.

  - The *Brain dataset*, seen in Figure 5.8(a), is a noisy and low contrast MRI scan of a brain. The data was retrieved from [44]. The figure shows a midpoint (see
Figure 5.8(b)) and the final version (see Figure 5.8(c)) of the brain after a combination of 12 contrast enhancement, brightness enhancement, gamma correction, and denoising steps. The final image not only removes noise, but it highlights important features, such as the skull, white matter, and grey matter. The persistence and persistence-volume diagrams are again shown for comparison.

The *Lenna Grayscale Dataset*, seen in Figure 5.9 shows a series of seven edits to a noisy version of the classic Lenna dataset.

- **Color Images:** To demonstrate the effectiveness of our approach on color images, we apply topology-based edits on two images from [36].
  
  - The *Notre Dame dataset*, shown in Figure 5.10(a) is a photo of the Notre Dame Cathedral, where the foreground is underexposed. For this dataset, we first performed a (virtually invisible) brightness enhancement to the green color channel, to make the foliage a green hue (see Figure 5.10(b)). Next, we performed a series of two enhancements (denoising and contrast enhancement) to the saturation channel of the HSB colorspace (see Figure 5.10(c)). Finally, three enhancements (denoising, gamma correction, and contrast enhancement) were applied to the brightness channel of HSB. Persistence and persistence-volume diagram are not shown because of the number of diagrams involved (10 total—two per red, green, blue, saturation, and brightness).
  
  - The *Swan dataset*, shown in Figure 5.12(a) is a photograph of a swan with a mix of light and shadow. We perform a series of enhancements that include denoising of the brightness channel (see Figure 5.12(b)); followed by brightness enhancement in the saturation channel (see Figure 5.12(c)); and finally contrast enhancement and gamma correction of the brightness channel (see Figure 5.12(d)).
Figure 5.7. Evaluation on Florala dataset. The (a) original data is (b) cleaned by a combination of topology-based denoised, contrast enhancement, and gamma correction.
Figure 5.8. Evaluation on Brain dataset. Example of a brain MRI cleaned using a combination of all four functions.
Figure 5.9. Evaluation on Lenna Grayscale. (a/b) Original and with noise, (c-h) Series of denoising and contrast enhancement.

The Lenna Color Dataset, seen in Figure 5.11 shows a series of 11 edits to a noisy version of the classic color Lenna dataset.

5.5 Conclusions

In this chapter, we have presented a new approach to image enhancement based upon the topology of an image. Our approach provides a high-level of control to users, while not requiring an extensive number of interactions to achieve desirable results.

Like with most other image enhancement algorithms, artifacts are important concern. Our approach does not introduce new artifacts, per se, but instead it may emphasize existing image artifacts. For example in Figure 5.4, blocking artifacts appear due to lack of detail for generating a smooth result. In Figure 5.4, artifacts occur due once again to missing contrast and detail that lead to small differences in intensity ending up emphasized.
Figure 5.10. Evaluation on Notre Dame dataset. (a) Photograph of the Notre Dame cleaned using (b) the green color channel, then (c) the saturation channel, and finally (d) the brightness channel.
Figure 5.11. Evaluation on Lenna Color dataset. (a/b) *Lenna Color* Original and with noise added. (c-g) A series of denoising, contrast enhancement, and brightness enhancement steps.
Figure 5.12. Evaluation on Swan dataset. (a) Photograph of the swan enhanced using (b) denoising of the brightness channel, then (c) brightness enhancement in the saturation channel, and finally (d) contrast enhancement and gamma correction in the brightness channel.
Chapter 6: Application of Mapper in Point Cloud-based 3D Printed Objects

This chapter\(^5\) provides an application of Mapper in quality assessment of 3D printed models.

6.1 Introduction

3D printing is gaining incredible popularity in low-yield manufacturing for customized or specialized parts. However, assessing the quality of models before they are printed remains a challenging problem \[74\], particularly when you consider point cloud-based models \[55\], such as those that come from 3D scanners. This chapter introduces an approach to quality assessment, which uses techniques from TDA to compute a topological abstraction of the eventual printed model and the empty space around and contained within it. This abstraction enables investigating certain properties of the model, with respect to print quality, and identifies potential anomalies that may appear in the final product.

6.2 Mapper and Persistent Homology

This approach uses two of the fundamental tools of TDA, namely Mapper \[70\] and persistent homology \[35\], to provide users with feedback about their models (see Figure 6.1). Mapper is used in two ways. First, it is used to extract information about the layer-by-layer connectivity of the model to be printed, providing an abstraction of the overall shape of the object. Second, it is used to determine the topology of the empty space contained within

\(^{5}\)Part of this chapter was published in Computer-Aided Design and Applications (2019) \[65\]. Permission to reproduce in the dissertation is included in Appendix A.
and surrounding the printed model. Persistent homology on the other hand is a tool that normally is used to provide a multiscale view of connected components, holes/tunnels, and voids in data of any dimension. Our approach uses persistent homology for the detection of connected components and holes within a printed layer.

The inner workings and associated details of both Mapper and persistent homology are quite complicated, and so we refer the reader to Chapter 2 and prior work for a better understanding [35, 70]. We will instead provide an intuition about the types of structures captured by each of these tools.

![Multilayer Filled Space Topology](image1)

![Single Layer Topology](image2)

![Multilayer Empty Space Topology](image3)

Figure 6.1. Multilayers filled/Empty space topology. Our approach uses Mapper to look at the filled space topology of multiple layers (left) and empty space topology of multiple layers (middle). It uses persistent homology to understand the topology of a single layer (right).

6.2.1 Mapper

Mapper is a TDA tool that provides a graph-based abstraction of the topology of a mesh or point-based data. Mapper construction starts by first parameterizing and slicing the data. In our case the parameterization is vertical.

The graph vertices are created from connected components identified within each layer. In other words, the connected components of the layer are “collapsed” into graph vertices. There are many variations on identifying connected components from points. We use the persistent homology approach, introduced in the next subsection.

Finally, graph edges are added between components that touch on neighboring layers. This connection is made by adding a small amount of overlap to each layer. If one or more
points in the overlap region are contained within connected components from two different layers, those component vertices receive a graph edge. The resulting graph can describe the overall topology of the connected components of a printed object.

![Figure 6.2. Example of Mapper on a mesh. The (a) model is (b) sliced. (c) Connected components are collapsed to vertices and edges added for components that touch. (d) Finally, an illustration of the printed object is shown.](image)

Figure 6.2 shows an example of Mapper on a simple domain. First, (a) the input model is (b) sliced with layer thickness being set to equal the 3D printer’s layer resolution. Next, (c) the connected components are found and edges added when they touch. (d) Finally, the illustration of the printed object is shown for comparison. The nodes of the Mapper graphs do not provide any insight into the size or shape of a given connected component. Instead they provide insight into which components touch and how those components may or may not form holes in the output model.

Calculating the Mapper graph on the empty space is a similar process. However, to calculate the graph, the empty space first needs to be filled. This is done by populating the empty space with points. Then, Mapper construction proceeds identically on the empty space points. The approach is illustrated in Figure 6.3.

The calculation of Mapper is relatively inexpensive. The slicing operation is linear in the number of points. The connected component detection is naively quadratic in the number of
Figure 6.3. Example of Mapper on the empty space of a mesh. The (a) model has its empty space filled with points and is (b) sliced. (c) The connected components are collapsed to vertices and edges added for components that overlap. For illustration purposes, the vertices here are colored green for outside and purple for inside the model.

points per layer, but this can be improved with spatial partitioning. The overall performance can be improved by using a parallelized algorithm [41].

6.2.2 Persistent Homology

Given a topological space $X$, the homology groups $H_0(X)$, $H_1(X)$, and $H_2(X)$, describe the connected components, holes/tunnels, and voids of the space, respectively. For example, consider the annulus in Figure 6.4(a). It has a single connected component. It also has a single hole/tunnel through the middle. Finally, it contains no void.

The multiscale notion of homology, called persistent homology, extracts the homology groups of a set of points considering different resolutions. A topological feature therefore has a minimum resolution where it first appears, known as the birth time, and a maximum resolution it is still visible, known as its death time. This can be intuitively thought of as the thickening of points. Figure 6.4(b-e) shows an example. Starting with (b) 12 points, the
Figure 6.4. (a) An annulus. (b-e) Example of persistent homology as it relates to a point-based annulus. As points are thickened, from (b) to (e), a hole/tunnel forms in (c) and closes in (e).

points are thickened, until (c) they form a single connected component with a hole. As the points continue to thicken (d) the hole remains visible, until (e) the thickness of the points closes it.

The performance calculating $H_0$ connected components is the same process per layer as with Mapper, naively quadratic. Finding the $H_1$ homology groups (i.e. holes/tunnels) in persistent homology is quite expensive. This calculation builds a simplicial complex on the data in the form of a boundary matrix and performs a reduction, similar to Gaussian elimination, which leads to a worst case performance that is cubic in the number of points. The average run time is linear with a large time constant. We mitigate this by pre-extracting per-layer connected components and running this calculation only on those components.

6.2.3 Link Between Mapper and Persistent Homology

The most direct link between Mapper and persistent homology is to use the persistent homology approach in the calculation of $H_0(X)$ homology groups (i.e. connected components) for the individual slices of the Mapper algorithm. However, we augment the conventional Mapper implementation by further attaching the $H_1(X)$ homology groups (i.e. holes/tunnels)
to the individual nodes of the Mapper graph. By doing this, the number of holes in each
connected component is retained for further analysis.

6.3 The Topology of 3D Printing

It turns out that both Mapper and persistent homology have direct applications to 3D
printing anomaly detection. For Mapper, the slicing operation has a direct corollary in the
layers of a 3D printer. Therefore, the slice thickness, known as the cover, can be set to the
same value as the thickness of a single layer on the 3D printer (i.e. the z resolution). For
persistent homology, the calculation of connected components is the same as a physically
connected components within a single layer. The holes within each layer represent the holes
within the model. These can be determined by targeting the xy resolution of 3D printer of
interest. Furthermore, using the empty space, Mapper can provide information about the
watertightness of the model.

6.3.1 Visualization

Once the topology of the point cloud has been calculated, we provide a visualization
for inspecting the data. The visualization contains four components. The first, and most
important, is the Mapper graph of the printed model, as seen in Figure 6.5(a). The Mapper
graph nodes shows the individual connected components of the model. In addition, each
tunnel going through the connected component is represented by a red point in the node
visualization. The next visualization, as seen in Figure 6.5(d) is the Mapper graph calculated
on the empty space of the model, instead of the filled space. The last two visualizations are:
the 3D point cloud (see Figure 6.5(b)), with regions highlighted based upon the selection
of Mapper graph nodes, and a 2D slice visualization (see Figure 6.5(c)), again based upon
nodes selected in the Mapper graph.
Figure 6.5. Our software with the Stanford Dragon dataset. (a) The filled space topology is shown as a Mapper graph with holes denoted as red dots. (b) A 3D view and (c) a single slice view are shown for detail. (d) The empty space topology is shown only as the Mapper graph.

6.4 Results

We implemented our approached using a number of tools. First, data is converted into a point by any method of choice, such as [55]. In our case, PLY or STL files had their vertices extract directly. Our Mapper implementation is in Java. The software loads a point cloud, slices it, detects connected components, and exports the Mapper graph and connected component points for both the filled space and empty space. Each filled space connected component is then fed into Ripser\(^6\) for persistent homology detection of holes/tunnels. For the visualization of the Mapper graph, the layout was calculated using Graphviz\(^7\). The data was then fed into our visualization tool built using Processing\(^8\).

\(^6\)Ripser: https://github.com/Ripser/ripser
\(^7\)Graphviz: https://www.graphviz.org/
\(^8\)Processing: https://processing.org/
Figure 6.6. Results of Dragon dataset. The Mapper graph of the filled space (left) has 5 different portions (a-e) highlighted (right).

We tested our approach on the Dragon dataset from the Stanford 3D Scanning Repository. We used the points from the reconstructed dataset, which contained approximately 437,000 points. The question we were after was, if someone was to try to rasterize these points directly for 3D printing (ignoring any mesh connectivity), what sort of anomalies would occur. We first scaled the model to a height of 10 cm. We then chose the z resolution to be 3.3 mm and xy resolution to be 1.0 mm.
6.4.1 Original Model

After running our pipeline, the results are displayed in Figures 6.5 and 6.6. In Figure 6.6, the tree on the left overviews the entire structure of the graph. We will concentrate on the few circled regions.

First, starting with Figure 6.6(c) in yellow, notice that this region represents a portion of the body of the dragon. In this region, each ring forms a single connected component, each with a single hole through the middle. That is until the topmost ring, where a single connected component has two holes, beginning the bifurcation of the upper front and middle portions of the body, as seen in Figure 6.6(a) in dark blue. This feature can be observed in the graph by looking at the top most node in the yellow circle. Notice two red dots, indicating two holes in that component.

Next, notice the region Figure 6.6(b) in orange. In this region, the model itself splits and comes back together leaving a hole between the torso and tail. This can be observed in the graph as well. Starting after the bottom node of orange region, the graph bifurcates, indicating a split in the connected components, and merges again at the top. This splitting and merging pattern is indicative of an exterior hole in the model. This same type of splitting and merging behavior can also be noticed in the graph region circled in green and associated with Figure 6.6(d). This hole is caused by the leg and body coming together. However, it is difficult to observe by looking at the 3D imagery of the point cloud. In fact, we could not find a good viewing angle that showed this hole directly.

We now look at the bottom slice of the model in Figure 6.6(e) in light blue. Looking at the graph, one may observe two nodes on the bottom layer that have many red points in the visualization. Each point representing a hole in the layer. This may represent a problem for watertightness, particularly given that this is the bottom layer. Observing the connected components represented by those two node in Figure 6.6(e), many holes are visible in the
layer due to inadequate resolution of the points. The initial concern about watertightness remains, given that these holes are not covered by a subsequent layer. Finally, the lack of watertightness can be confirmed by looking at the empty space graph in Figure 6.5(d). In this graph, there is a single component representing all empty space. If the model were watertight, at least two empty space components would form, one outside the model and one or more inside.

Figure 6.7. Error Corrected Results of Dragon dataset. (a) The filled space shows a single connected component and holes only on the interior. (b) The empty space has two connected components, (left) the outside of the model and (right) the inside of the model. This indicates that the model is now watertight.
6.4.2 Error Corrected Model

As a comparison, we have computed an error free version of the dragon model. To do this, the triangle mesh provided with the model was subdivided to calculate additional vertices until the point model became watertight. The result of the Mapper and persistent homology calculations can be seen in Figure 6.7. This new model contained 441,713 points (less than 1% increase from the original), making it visually indistinguishable from the original.

In Figure 6.7(a), the Mapper graph of the filled space looks identical to the Mapper graph of the original in Figure 6.6(a). The persistent homology calculation however is quite different. The number of red dots (i.e. holes in the model) have reduced significantly. In fact, the only holes that remain are those representing the major empty cavities of the model’s interior.

In Figure 6.7(b), the Mapper graph of the empty space is shown. The most important aspect of these new graphs is that there are now two connected components. Figure 6.7(b)(left) represents the connected component of the air surrounding the model. Figure 6.7(b)(right) represents the air inside the model. The lack of connection between these two components indicates that the model is now watertight.

6.4.3 Runtime Performance

We tested the runtime performance of our analysis on the Dragon dataset by varying the three main parameters, the number of slices, slice overlap, and the xy grid resolution. The results can be seen in Figure 6.8. These results show that persistent homology is almost always the largest cost. This high cost can be attributed to regions that have large connected components.
Figure 6.8. Performance Evaluation. Performance result varying the three main parameters of the approach: (a) number of slices, (b) slice overlap, and (c) xy grid resolution. In all results, the time for slicing is presented in milliseconds, while Mapper and persistent homology are reported in seconds.

6.5 Conclusions

In conclusion, in this chapter we presented an approach for using TDA in the evaluation of the quality of 3D printed objects using point cloud-based models. We made some simplifying assumptions. For example, we assume that 3D printing resolution is uniform across the entire xy domain, which is not necessarily true. We also chose a naive rasterization procedure, though any other pre-rasterized model would be adequate for analysis in this pipeline.
It is also important to note that this approach, as presented, does not report specific problems, aside from watertightness. It instead enables a number of qualitative analyses that depend upon a user’s expectation for the output of their model, including certain global or regional problems, such as issues with number of tunnels expected per component; whether the tunnels are connected; the number of connected components per slice; and which connected components make contact slice-to-slice. This essentially enables answering the question, “does the printed model topology match my expectations?”
Chapter 7: Conclusion and Future Work

7.1 Summary

The field of Topological Data Analysis is advancing rapidly in theory, algorithms, and applications. In this dissertation, we have made several contributions to the advancement of algorithms and applications of TDA as follows.

In terms of efficient algorithms, we integrated three techniques to develop a new hybrid solution to the parallel computation of merge trees, one of the integral of building blocks of contour trees and Reeb graphs, in Chapter 3. In Chapter 4, we developed and implemented two efficient algorithms for the critical points pairing in Reeb graphs (and by consequence join, split, and contour trees) based on feature persistence in TDA.

In terms of TDA applications, in Chapter 5, we designed and implement a prototype software for color image enhancement based on using topological features from the contour tree and persistence diagram. In Chapter 6, we presented an application of Mapper and persistent homology on quality assessment of point clouds based 3D printed objects.

7.2 Vision

Although there has been great progress made in the field of TDA, there are many research directions to follow this dissertation, as shown in the pipeline in Figure 1.1.

One pressing issue continuing to hindering the wide application of TDA is the lack of efficient and parallel algorithms in many computations in TDA, such as Reeb graphs construction, persistent homology computation, and Mapper construction. Along with the initial
work on the parallelization of merge tree computation in Chapter 3, there are other portions of the contour trees construction that need to be parallelized to be more useful in practical application, such as image enhancement in Chapter 5. For example, parallelizing the union of merge trees to form contour trees, calculating of persistence diagram, and subtree selection in augmented contour tree would all be essential to real world applications.

Considering the relatively widespread of Topological ToolKit developed by Tierny et al. [75], it would be beneficial for both TDA research groups and general scientific visualization community to integrate our algorithms on parallel merge trees from Chapter 3, critical point pairing of Reeb graphs in Chapter 4, and aspects of image enhancement from Chapter 5 into Topological Toolkit.

Another big thrust in the community of TDA is the combination of machine learning and TDA [53]. In Chapter 4, we developed two algorithms to compute the persistence feature of Reeb graphs, how to effectively use them in computer vision [82] and combine persistence diagram of Reeb graphs with deep learning and shape analysis would potentially be quite rewarding.
References


Algorithms and Software for Computational Topology.


Appendix A: Copyright Permissions

The permission below is for the reproduction of material in Chapter 3 and Chapter 6, from Computer-Aided Design and Applications, CAD SOLUTIONS LLC.

**AUTHOR RIGHTS**

3. You hereby assert your moral rights to be identified as the author of the Article according to the UK Copyright Designs & Patents Act 1988.

4. You are permitted to use the material in the ways described in the Schedule of Author’s Rights providing that you meet all the conditions set out in the Schedule. These are rights which are personal to you and cannot be transferred by you to anyone else.

The permission below is for the reproduction of material in Chapter 4 and Chapter 5, from International Symposium on Visual Computing, Springer Nature Switzerland AG, Gewerbestrasse 11, 6330 Cham, Switzerland

**§ 2 Rights Retained by Author**

Author retains, in addition to uses permitted by law, the right to communicate the content of the Contribution to other research colleagues, to share the Contribution with them in manuscript form, to perform or present the Contribution or to use the content for non-commercial internal and educational purposes, provided the original source of publication is cited according to the current citation standards in any printed or electronic materials. Author retains the right to republish the Contribution in any collection consisting solely of Author’s own works without charge, subject to ensuring that the publication of the Publisher is properly credited and that the relevant copyright notice is repeated verbatim. Author may self-archive an author-created version of his/her Contribution on his/her own website and/or the repository of Author’s department or faculty. Author may also deposit this version on his/her funder’s or funder’s designated repository at the funder’s request or as a result of a legal obligation. He/she may not use the Publisher’s PDF version, which is posted on the Publisher’s platforms, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgment is given to the original source of publication and a link is inserted to the published article on the Publisher’s website. The link must be provided by inserting the DOI number of the article in the following sentence: “The final authenticated version is available online at [https://doi.org/][Insert DOI].” The DOI (Digital Object Identifier) can be found at the bottom of the first page of the published paper.