

W-Structures in Contour Trees

Petar Hristov

**Submitted in accordance with the requirements for the degree of
Mathematics and Computer Science MSc**

Session 2016/2017

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Project Report	Report	SSO (12/09/2017)
Implementation	Software code and URLs	Supervisor, Assessor (12/09/2017)

Type of project: Theoretical

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) _____

Summary

As the sheer amount of data that is collected in business and scientific applications reaches exascale it is becoming an increasingly challenging task to process it and extract useful information. To do so we not only need efficient algorithms that work over distributed and multicore hardware but also ways of reducing its volume.

Topological Data Analysis is a general framework for analysing and comparing the significance of subsets of data. This allows us to process only the most significant ones and greatly reduce computational overhead. Developing algorithms in this field is a challenging task because of the ballancing act one must do between continous mathematical models and discrete computational models.

A principal tool in topological data analysis that is used extensively in scientific visualisation is the contour tree. It is a discrete data structure that represents the topological structure of a scalar field. The aim of this dissertation is to lay the foundations of understanding a particular pathological edge case that emerges in the state of the art parallel algorithm for contour tree computation and to explore its connection to other fields of topological data analysis.

Acknowledgements

First and foremost I would like to thank my academic supervisor Dr. Hamish Carr. This work would not have been possible without his guidance and encouragement. I would also like to thank my girlfriend Hristina Tsoleva for her support and my good friends Atanas Angelov and Ishan Lee.

Contents

1	Introduction	3
2	Background	5
2.1	Point Set Topology	5
2.2	Differential Topology	7
2.2.1	Reeb Graph	8
2.3	Algebraic Topology	9
2.3.1	Simplicial Complexes	9
2.3.2	Euler Characteristic	10
2.4	Graph Theory	11
2.4.1	General Graph Theory	11
2.4.2	Tree Diameter Algorithms	11
3	Contour Trees	13
3.1	Typical Input Data	13
3.2	Contour Tree Computation	14
3.3	Height Trees	14
3.4	Serial Algorithm	15
3.5	Parallel Algorithm	18
3.6	Contour Tree Simplification	19
4	W-structures - Theory and Algorithms	23
4.1	Formal Description of the w-structures	23
4.2	Linear Time Algorithm - 2xBFS	26
4.2.1	Pathological Cases in 2xBFS	31
4.2.2	Attempts at resolving the accuracy of 2xBFS	32
4.3	Dynamic Programming Algorithm - DP	32
5	Homology	41
5.1	Homology	41
5.2	Reduced and Relative Homology	45
5.3	Inclusion Maps and Induced Maps on Homology	47
5.4	Persistent Homology	48
5.5	Extended Persistence	51
6	Extended Persistence and Branch Decomposition	53
6.1	Persistence of Branches	53
6.2	Persistence Pairs vs Branch Decomposition Pairs	54
7	Empirical Study	61

7.1	Algorithm Implementations	61
7.2	Data sets Overview	62
7.3	W-detector Algorithms	62
7.4	Dataset w-diameter analysis	65
8	Conclusion	67
8.1	Personal Reflection	67
8.2	Future Work	68
	References	70
	Appendices	75
A	Ascending Filtration of the Contour Tree	77
B	Descending Filtration of the Contour Tree	79
C	Additional Proofs	81
D	Github Repositories	83
E	Ethical Issues Addressed	85
E.1	Data Sources	85
E.2	Software	85

List of Figures

2.1	Simplices of dimension 0, 1, 2 and 3.	9
2.2	A simplicial complex.	10
3.1	Triangulation of input data to obtain a simplicial mesh.	13
3.2	The simplicial mesh, join and split trees and contour tree.	16
3.3	Hierarchical branch decomposition of the contour tree from Figure 3.2 b.	20
4.1	A path and its monotone path decomposition.	24
4.2	Two possible types of kinks (vertices are labeled with their height).	24
4.3	Relative position of vertices in Case 1.1 (dotted lines are paths).	27
4.4	Relative position of vertices in Case 1.2 (dotted lines are paths).	28
4.5	Relative position of vertices in Case 2 (dotted lines are paths).	30
4.6	Pathological cases in the 2xBFS algorithm (dotted lines are monotone paths of length at least three).	31
4.7	Infinite cycle between u and v (dotted lines are monotone paths of length at least 3).	32
5.1	A simplicial complex.	41
5.2	Example of a filtration of a simplicial complex.	48
5.3	Examples of visualising persistent homology of the filtration on Figure 5.2.	50
6.1	Branch decomposition of the join tree.	55
6.2	Branch decomposition of the split tree.	55
6.3	Branch decomposition of the contour tree.	56
6.4	Ascending filtration of the dataset.	57
6.5	Descending filtration of the data set.	58
6.6	Barcode diagrams of the persistent pomology of the ascending and descending filtration.	58
6.7	Branch decomposition of the join/split trees and extended persistence of the filtrations.	59
7.1	Running time of NxBFS on randomly generated trees.	63
7.2	Running time of 2xBFS (blue) and DP (red) on randomly generated trees.	64
A.1	Ascending filtration of the contour tree from Figure 3.2 b.	77
B.1	Descending filtration of the contour tree from Figure 3.2 b.	79

Chapter 1

Introduction

The mathematical field of topology studies the qualitative properties of geometric objects. It is the natural field to study for example the "shape" of a surface by decomposing it into path-connected components. While topology has traditionally been only considered within pure mathematics new methods allow for topological properties of data to be computed in practice. These new methods are the study of Computational Topology [41] and Topological Data Analysis [11]. These emerging fields on the edge of pure mathematics and computer science leverage theory from topology to produce algorithms for solving various problems in structural biology [39, 19], visualisation [10, 13, 15], medical imaging [36] and computer vision [9, 28].

In this dissertation we will be most interested in utilizing computational topology in the context of scientific visualisation. We shall do so with the use of a tool that has been well established in recent years called the contour tree [14]. The contour tree is a discrete graph data structure that is used to summarise the connectivity of planar cross sections of a scalar field. The utility of the contour tree is in that it can be used to identify and display the most topologically significant features in data with little to no human interaction. Such automated tools become invaluable when the amount of collected data far exceeds the capability of a human to process manually.

The central problem that we will discuss in this dissertation is a theoretical computational efficiency limitation of the current state of the art algorithm for data-parallel contour tree computation [17]. The cause of this issue are certain substructures of contour trees we call w-structures. They hinder parallel performance by serialising parts of the computation [16]. Our goal is to understand how and why these w-structures appear in contour trees of real life data. We will accomplish this by developing new algorithms that detect the existence of w-structures and extract them for further study.

The second problem we will address is that of contour tree simplification [35]. Contour tree simplification is the process of reducing the size of a contour tree by removing parts of it that correspond to topologically insignificant features of data. We will analyse the process of contour tree simplification using a more general tool from topological data analysis called persistent homology [21]. We will pose and answer the question of whether the two approaches are equivalent. A counterexample based on the w-structures we have introduced will make clear that fact that they are not.

The material in this dissertation is spread throughout eight chapters. The second chapter provides the reader with the necessary mathematical background to tackle most of the rest of the dissertation. Chapter three introduces the concept of contour trees and the state of the art algorithms found in the literature for computing them. Chapter four constitutes the first part of our original research. We explore the theoretical properties of w-structures and develop three algorithms for detecting and extracting them from contour trees. In chapter five we take a step

back to introduce more mathematical background that would enable us to address our second goal of comparing contour tree simplification and persistent homology. In this chapter we will cover the basics of a subfield of Algebraic Topology called Homology and introduce the theory behind Persistent Homology. Chapter six is the second part of our original research. We explore the connection between contour tree simplification and persistent homology by computing and comparing the output of both on a specifically chosen data set. In chapter seven we present an empirical study on the w-structures by implementing and analysing the algorithms we created in chapter four and running them on both artificially generated and on real life data sets. We will use those algorithms to demonstrate that the w-structures do appear in real life data sets and that the theoretical algorithmic issues they cause translate to issues with practical computational performance.

Chapter 2

Background

The two key concepts we will introduce in this dissertation are the Contour Tree and Persistent Homology. In order to be able to do this we have to first take a step back and walk the reader through a range of other mathematical disciplines. The preliminaries include Point Set Topology, Differential Topology, Algebraic Topology and Graph Theory. We will opt for introducing these fields with a more practical and computational flavour and provide the reader with both the necessary formalism and intuition behind the main definitions and results we will use in the following chapters.

2.1 Point Set Topology

The first branch of Topology that we will introduce is Point Set Topology. It forms the underlying framework on top of which mathematicians build the concepts of continuous spaces and functions. Point Set Topology is the study of sets that possess certain mathematical structure. The core concept in Point Set Topology is the mathematical structure known as the topology of a set. The topology of a set makes the notion of whether two elements of a set are "close" or "near" to one another rigorous. Elements of a set which are close or near to one another are said to be a part of an open set. In this chapter we will borrow definitions and results from one of the standard introductory topology textbooks [31].

Definition 1. *Let X be a set and τ be a set of subsets of X . The set τ is a topology on X when the following holds:*

- X and $\emptyset \in \tau$.
- If U and $V \in \tau$ then $U \cap V \in \tau$.
- If $\{U_\lambda\}_{\lambda \in \Lambda}$ is a family of subsets of X , where $U_\lambda \in \tau$ for all $\lambda \in \Lambda$, then $\bigcup_{\lambda \in \Lambda} U_\lambda \in \tau$.

We will call the elements of X points and the elements of τ open sets or simply open. An open set is an open neighbourhood of a point when the point is in the open set. We must stress that the topology we endow on a set is by no means unique. For example if X is any set then one valid topology on X may consist of all subsets of X while another may simply be $\{\emptyset, X\}$.

Let us now introduce the topology we are going to use on n -dimensional Euclidean space or simply \mathbb{R}^n . It is called the standard topology and it is based on the standard definition of distance between points in \mathbb{R}^n . Let $x = (x_1, x_2, \dots, x_n)$ be a point in \mathbb{R}^n . We can define the open ball around x of radius ϵ as $B_\epsilon(x) = \{y \in \mathbb{R}^n : d(x, y) < \epsilon\}$ where we define the distance function as $d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$. The standard topology on \mathbb{R}^n consists of the open balls around all points of all possible radii and their finite intersections and arbitrary unions.

Next we will define a special class of functions that preserve the properties of topological spaces. Those are the continuous functions.

Definition 2. A function $f : X \rightarrow Y$ is said to be continuous when the preimage of an open set in Y is an open set in X .

In formal notation if $U \in Y$ is open in Y then $f^{-1}(U)$ is open in X . This definition captures the understanding we have of continuity from calculus in the more general context of topology. If f is a bijection and f^{-1} is also continuous we will call f a homeomorphism. Homeomorphisms play a special role in topology. Two topological spaces are homeomorphic when there exists a homeomorphism between them. As continuous functions preserve open sets it follows that homeomorphic spaces are topologically identical. This is the reason why topologists are mostly interested in classifying and analysing spaces up to homeomorphism.

We will call a property of a space that is preserved under homeomorphisms a topological invariant. The first topological invariant we will introduce is path-connectedness. Path-connectedness is based on paths between points in a topological space.

Definition 3. Let X be a topological space and let $x, y \in X$ be any two points. A path between x and y in X is a continuous function $f : [0, 1] \rightarrow X$ such that $f(0) = x$ and $f(1) = y$.

Using this definition we can define a path-connected topological space as follows.

Definition 4. A topological space X is said to be path-connected if there exists a path between any two points $x, y \in X$.

Topological invariants play a crucial role in differentiating between topological spaces. Since path-connectedness is a topological invariant the continuous image of a path-connected topological space is also path-connected. It follows that there cannot exist a homeomorphism between a topological space that is path-connected and one that is not.

We will now introduce two different ways in which you can obtain new topologies from already known topologies. The first way is known as the subspace topology. A subspace of a topological space X is any subset of points in X .

Definition 5. Let $A \subseteq X$ be a subspace of X . We define the open sets for a topology on A as the intersection of the open sets in X with A .

This means that a set $U \subseteq A$ is open in A exactly when $U = U' \cap A$ where U' is open in X . This result allows us to obtain a topology of A by taking all possible intersections of the open sets in X with A . For example let $X = \mathbb{R}$ and $A = [0, 1]$. The set $[0, 1/2) = (-1/2, 1/2) \cap [0, 1]$ is open in A because $(-1/2, 1/2)$ is open in X . This does not imply that $[0, 1/2)$ is open in the topology of X , only in the topology of A .

The second way of obtaining new topologies is via the quotient topology. To understand it we must first define a quotient space via an equivalence relation. An equivalence relation \sim defined on a topological space X partitions all points in X into equivalence classes. The equivalence class of a point $x \in X$ is the set $[x] = \{y \in X : x \sim y\}$. The set of all equivalence classes is called the quotient of X by \sim and denoted as X/\sim . Let also $\pi : X \rightarrow X/\sim$ be the map that takes a point x of X to its equivalence class in X/\sim .

Definition 6. Let X be a topological space and \sim be an equivalence relation defined on X . The

quotient topology of X/\sim is formed by the sets $U \subseteq X/\sim$ such that $\pi^{-1}(U)$ is open in X .

By this definition the function π is continuous. We can use this fact to infer that if a topological space X is path-connected then X/\sim is path connected for any equivalence relation \sim because there exists a continuous function $\pi : X \rightarrow X/\sim$. An example of this is when we take quotients of subsets of topological spaces. Let $A \subseteq X$. We can define an equivalence relation as $x \sim y$ whenever both $x, y \in A$. We will call the resulting quotient space X/A . The geometrical interpretation of X/A is that all points in A are contracted to a single point in X/A . As a direct example of this consider the closed disk $D = \{x^2 + y^2 \leq 1\}$ and its subset the circle $S^1 = \{x^2 + y^2 = 1\}$. The quotient space D/S^1 is homeomorphic to the three dimensional sphere $S^2 = \{x^2 + y^2 + z^2 = 1\}$.

Now we will present our final definition. That of a topological manifold - a mathematical generalisation of a surface.

Definition 7. A d -manifold is topological space where every point has an neighbourhood that is homeomorphic to \mathbb{R}^d .

An example of a 0-dimensional manifold is a single point in \mathbb{R}^n . Examples of one dimensional manifolds are lines, circles, graphs and curves. Examples of 2-dimensional manifolds are the surfaces we are familiar with from geometry such as the sphere, the torus and so on. In this dissertation we will only consider manifolds of dimension zero, one, two and three. The reason for this is that our primary focus is on the visualisation aspect of computational topology. Visually modelling natural spacial phenomena prohibits us from using high dimensional manifolds because they cannot be properly embedded in two or three dimensional Euclidean space.

It is often difficult to analyse the topology of a space by just considering its open sets. This is why in the following two sections we will employ additional tools from other fields of mathematics to aid in our analysis of the topology of a space. These tools are differentiable functions over differentiable spaces and combinatorial approximations of topological spaces.

2.2 Differential Topology

Differential topology is the study of differentiable functions defined on differentiable manifolds. One of the most well developed fields of differential topology is Morse Theory [30, 32]. Morse theory is the study of the relationship between spaces and functions defined on them. One way we can study manifolds via differentiable functions is by analysing the critical values of the functions. Due to complexity of doing so especially in relation to degenerate critical points we will restrict ourselves to a special class of differentiable functions called Morse functions.

Definition 8. A function $f : M \rightarrow \mathbb{R}^n$ is a Morse Function if f is smooth and at critical points the Hessian (matrix of second partial derivatives) is full rank.

In order to analyse the topology of scalar fields will restrict our attention even further and consider Morse functions whose codomain is \mathbb{R} . Points of M whose first derivative is zero are called critical points. All other points of M are called regular.

We can use a Morse function defined on a manifold to decompose it into a family of path-

connected subsets. We can analyse the subsets to obtain global topological information about the connectivity of the manifold. Examples of such families of subsets are level sets, sublevel sets and super level sets.

Definition 9. A level set at a value h of a Morse function $f : M \rightarrow \mathbb{R}$ is the set $f^{-1}(\{h\}) = \{x \in M : f(x) = h\}$

Sublevel sets are defined in terms of the preimage of f of intervals of the form $[-\infty, a]$. A sublevel set at a is defined as $f^{-1}([-\infty, a]) = \{x \in M : f(x) \in [-\infty, a]\}$. Superlevel sets are defined analogously in terms of intervals of the form $[a, \infty]$.

Morse functions ensure the following properties hold:

- None of the critical points are degenerate.
- Changes in the topology of level sets, sublevel sets and superlevel sets only happen at critical points.
- A Morse function defined on a surface has a finite number of critical points.

Morse functions allow us to decompose a manifold into its level sets. We will use the theory we have developed so far to introduce the principal tool that allows us to analyse how the connectivity of level sets $f^{-1}(\{h\})$ changes as we vary the input parameter h .

2.2.1 Reeb Graph

The Reeb Graph is a tool that encapsulates the evolution of the topology of level sets of a continuous function. When the function is Morse, an edge in the Reeb graph corresponds to a sequence of contours in the level sets whose topology does not change. The vertices correspond to critical points where the topology of those components does changes. An example of a topological change is when connected components in the level sets appear or disappear or when two connected components split or merge. Morse theory ensures that critical points occur at distinct values of the parameter and are isolated. This removes ambiguities that may arise in the construction of the Reeb graph.

Definition 10. Given a topological space X and a continuous function $f : X \rightarrow \mathbb{R}$ we can define an equivalence relation \sim such that two points x, y in X are equivalent when there exists a path between them in a level set $f^{-1}(\{h\})$ for some $h \in \mathbb{R}$. The Reeb Graph is the quotient space X / \sim together with the quotient topology.

We can think of the Reeb graph of a space X as the quotient space where the connected components of all level sets are contracted to a single point. The resulting topological graph can also be thought of as a discrete graph. To do so we must enumerate the vertices and record all edges between them.

The reason we have defined Reeb graphs is because the contour tree is a special case of the Reeb graph. We will leave it as this and return to the topic in the beginning of the next chapter. Before doing so we must take a look at certain tools from Algebraic Topology that allows us to translate the continuous mathematical results we have obtained so far into the realm of finite combinatorial structures that would allow us to perform actual computation.

2.3 Algebraic Topology

Algebraic Topology is a branch of topology that uses tools from the field of abstract algebra to study topological spaces. The primary goal is to derive algebraic structures such as groups, rings and vector spaces from topological spaces that remain invariant under continuous mappings. Modern Algebraic Topology has its roots in combinatorially defined topological spaces [27]. Unlike Point Set Topology and Differential Topology this allows us to obtain exact algorithms for computing the algebraic invariants we are interested in. To make matters clearer we will introduce one of the most basic combinatorial topological spaces - Simplicial Complexes and then we will introduce one of the earliest discovered algebraic invariants - the Euler Characteristic. We will continue our discussion on Algebraic Topology in Chapter 5 where we will see how the concept of the Euler Characteristic can be generalised to the field of Algebraic Topology called Homology.

2.3.1 Simplicial Complexes

Simplicial Complexes are one of the first combinatorially flavoured topological spaces one encounters in Algebraic Topology. A simplicial complex is a subset of \mathbb{R}^n that consists of points, line segments, triangles and their higher dimensional analogues attached to one another in a single geometric object. In order to understand simplicial complexes we must first define their basic building blocks [22, p. 51].

Definition 11. Let $\{v_0, \dots, v_k\}$ be k points in \mathbb{R}^n . The convex combination of the points is the sum $\sum_{i=0}^k \lambda_i v_i$ where $\lambda_i \geq 0$ and $\sum_{i=0}^k \lambda_i = 1$.

If we decide to take the subset of \mathbb{R}^n covered by all possible convex combinations we obtain the convex hull of the points.

Definition 12. Let $\{v_0, \dots, v_k\}$ be points in \mathbb{R}^{k+1} . The convex combination of those points is the k -simplex defined by the points. We will write that simplex as $[v_0, \dots, v_n] \subset \mathbb{R}^{k+1}$

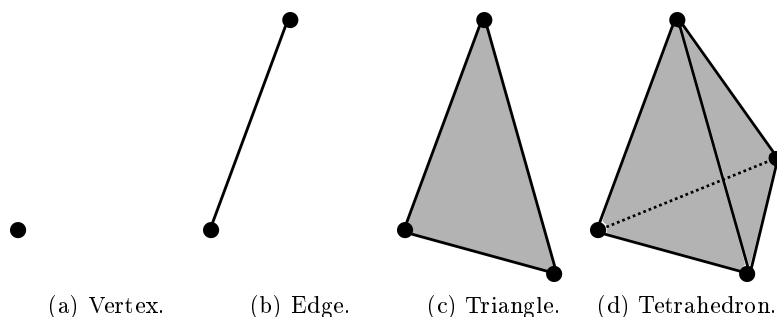


Figure 2.1: Simplicies of dimension 0, 1, 2 and 3.

The number k is also called the dimension of the simplex.

We will call the simplices of dimension 0, 1, 2 and 3 vertices, edges, triangles and tetrahedron respectively. As we have mentioned previous we are primarily interested in introducing mathematics as a tool to use in visualisation. Therefore we will have no use for simplices of higher dimension so will avoid naming them altogether.

A face of a simplex is the convex hull of a non-empty subset of its points. For example the faces of the tetrahedron are the four triangles, six edges and four vertices. To construct a simplicial complex all we have to do is take the union of a number of simplices and "glue" them together along common faces without allowing self-intersection.

Definition 13. A simplicial complex K is a finite collection of simplices such that if τ is a simplex in K then all faces of τ must be simplices in K . Furthermore the intersection of two simplices in K is either empty or a common face of both.

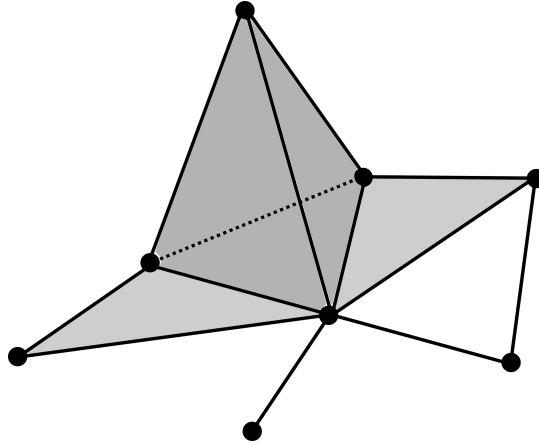


Figure 2.2: A simplicial complex.

We obtain the topology of a simplicial complex by embedding it in Euclidean space and considering its subspace topology as a subset. After formalising the concept of a simplicial complex we will introduce our first algebraic invariant.

2.3.2 Euler Characteristic

The first topological invariant of algebraic nature we shall encounter is the Euler Characteristic. It is denoted as χ and it assigns an integer to simplicial complexes through a generalisation of counting [25, p. 44]. The concept was originally defined for polyhedra as an alternating sum of the form $|V| - |E| + |F|$, where V is the set of vertices, E the set of edges and F the set of faces. The Euler Characteristic can be generalized to simplicial complexes as an infinite alternating sum 3-dimensional simplices, 4-dimensional simplices and so on.

$$\chi = k_0 - k_1 + k_2 - \dots = \sum_i (-1)^i k_i,$$

where all k_i is the number of i -dimensional simplices for $i \in \mathbb{Z}^+$ and $k_j = 0$ for j bigger than the dimension of the highest dimensional simplex in the simplicial complex.

The Euler Characteristic is a topological invariant. This allows us to compute the Euler Characteristic of topological spaces which are not simplicial complexes. Let us take for example the sphere. We will call any simplicial complex that is homeomorphic to the sphere its triangulation. The most basic triangulation of the sphere is the tetrahedron (excluding its volume). Therefore the Euler Characteristic of the sphere is $\chi = 4 - 6 + 4 = 2$.

2.4 Graph Theory

The last piece of background theory we will cover is from Graph Theory. In this dissertation we will make use of certain notation and algorithms that we will define here.

2.4.1 General Graph Theory

We will assume that the reader has is familiar with basic concepts from graph theory such as graphs, subgraphs, vertices, edges, paths and cycles and the basic algorithm such as Breadth First Search (BFS) and Depth First Search (DFS) [20]. For a graph $G = (V, E)$ we will use notation $V(G)$ for the vertices of G and $E(G)$ for the edges of G . We will use the notation $N(u)$ for the neighbourhood of u or all the vertices u is adjacent to. We will use the notation $d(u, v)$ where $u, v \in V(G)$ for the length of the shortest path between u and v in G .

Throughout this dissertation most of the graphs we will be working with will be trees. A tree is a connected graph that has no cycles. We will typically refer to trees as $T = (V, E)$. For our intents and purposes we shall define a subtree of a tree as a connected subgraph of a tree. We will refer to subtrees of rooted trees by their root. If T is a rooted tree and u is a node in T then T_u is the subtree of T whose root is u . In this notation if s is the root of T then $T_s = T$. If u is any vertex that is not the root of T then T_u is the (vertex-wise) maximal subtree of T that contains u but does not contain the parent of u . We remind the reader that $N(u)$ does not include the parent of u in a rooted tree.

In Chapter 4 we will be interested in finding the longest path in a tree. This is known as the diameter of a tree. Here we will describe two of the most well known linear time tree diameter algorithms.

2.4.2 Tree Diameter Algorithms

The first algorithm we will discuss is based on the following theoretical result [1].

Lemma 1. *Let s be any vertex in a tree. The most distant vertex from s is an endpoint of a tree diameter.*

To implement this algorithm we require a way of finding the most distant vertex from a given vertex. This can be done using Breadth First Search (BFS) [20, p. 594]. Let T be a tree and $s \in V(T)$ be any vertex. We can run BFS with s as its root to find a vertex u such that $d(s, u) \geq d(s, t)$ for all $t \in V(T)$. We can then run a second BFS with root u to obtain a vertex v such that $d(u, v) \geq d(u, t)$ for all $t \in V(T)$. Since u is the farthest vertex from s by Lemma 1 it must be the endpoint of a diameter. The diameter of T is the longest path in T . Therefore the second BSF produces a path whose length as much as the diameter of T . Therefore $d(u, v) \geq d(a, b)$ for all $a, b \in V(T)$.

The space and time complexity of BFS are linear and therefore the space and time complexity of this algorithm are linear as well. This follows from the fact that the algorithm consists of running BFS two consecutive times. For the implementational details we refer the reader to [2].

The second approach [3] is based on the Dynamic Programming paradigm. Dynamic programming is a method that is used to solve optimisation problems whose solution can be reformulated recursively through solutions of subproblems of the original problem. The key ingredients in developing a dynamic programming algorithm are [20, p. 359]:

1. Characterise the structure of the optimal solution.
2. Recursively define the value of the optimal solution.
3. Compute the value of the optimal solution.

We will characterise the structure of the optimal solution through all subtrees of a tree. If we root a tree T in any of its vertices then the structure of the optimal solution is defined via all the rooted subtrees of T or $\{T_u\}_{u \in V(G)}$. We can recursively define the value of the optimal solution with the following observation. Starting at the root the longest path in the tree either goes through the root or is entirely contained in one of the subtrees rooted at the children of the root. This reasoning can be extended to all rooted subtrees of the tree.

In order to formalise this idea we will make use of two functions. Let T be a rooted tree with root s . Let $h(u)$ be the height of the subtree rooted at u . The height is defined as the longest path in T_u from u to one of the leaves of T_u . We will also call such a path a height path. Let $D(u)$ be the length of the longest path contained entirely in T_u . The function D will contain the value of the optimal solution for all subtrees. The value of $D(s)$ is either equal to the value of $D(u)$ where u is a child of s or it is equal to combining the two maximum height paths of two of the children of u . This is summarised by the following formula.

$$D(s) = \max \left\{ \max_{u \in N(s)} \left(D(u) \right), \max_{\substack{u, v \in N(s) \\ u \neq v}} \left(h(u) + h(v) + 2 \right), \max_{u \in N(s)} \left(h(u) + 1 \right) \right\}.$$

The first term describes the case when the longest path is contained entirely in one of the subtrees rooted at a child of s . The second term combines the two longest height paths in two distinct children u, v of s and adds an additional 2 to account for the edges us and vs . The last term is given in the case where s has exactly one child and we cannot combine two height paths.

The base case for this recursive formula is at the leaves of T . If u is a leaf of T then $V(T_u) = \{u\}$. This allows us to set $h(u) = 0$ and $D(u) = 0$ and consider all leaves as base cases for the recursive formula. This algorithm can be implemented in linear time using Depth First Search (DFS) [20, p. 603] by using two auxiliary arrays that hold the values for $h(u)$ and $D(u)$ for every $u \in V(T)$. For the implementational details we refer the reader to [4].

Chapter 3

Contour Trees

The Reeb graph of a scalar field is connected and acyclic [22, p. 141]. As such we will call the Reeb graph of a scalar field the contour tree. In this chapter we will assemble the theory we have presented thus far and use it to introduce the state of the art serial and parallel algorithms for contour tree computation. We will begin with a short discussion on how we treat input data and what theoretical simplifying assumptions we are making. Afterwards we will present an overview of contour tree algorithms and then introduce some graph theoretical properties of contour trees. Next we will describe in detail how the serial and parallel contour tree algorithm work. This will lead us to defining the so called w-structures. We will demonstrate why exactly they are a pathological case that causes poor performance in the parallel contour tree algorithm. The final topic of this chapter is contour tree simplification. This is the process of identifying and removing parts of the contour tree that are not topologically significant.

3.1 Typical Input Data

Many scientific and medical applications require the sampling of scalar values from a bounded area or volume in two or three dimensional Euclidean space [12]. The theory we have presented so far is applicable only in the continuous setting but the resolution of any sampling process is finite. If we are to leverage this theory we must assume an underlying continuous function in the whole of the area or volume and not just at the sampled points. To do so we will construct an approximation of this function based on the values we have sampled. This is usually done by constructing a simplicial complex where the data points are the vertices and higher dimensional simplices are added to completely fill the space between them (see Figure 3.1). We will call the resulting data structure a simplicial mesh [12].

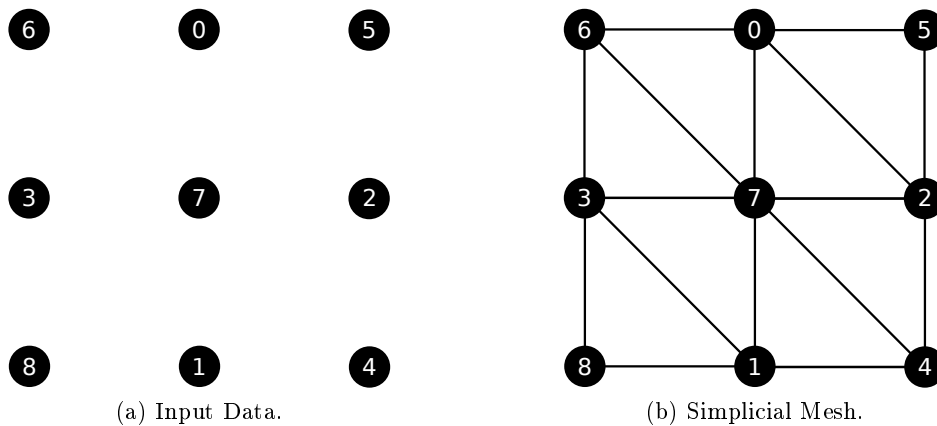


Figure 3.1: Triangulation of input data to obtain a simplicial mesh.

For simplicity and without loss of generality we will work with two-dimensional domains

where the value samples are evenly spaced out in a grid-like fashion (Figure 3.1). The values of the approximation function at the simplices are obtained via linear interpolation between the vertices of each simplex. As long as the original values we have sampled are unique it can be shown that the linear interpolation function is a Morse function and that all critical points are the vertices of the mesh [8].

3.2 Contour Tree Computation

The first efficient algorithm for constructing contour trees [38] is due to Van Kreveld et al. Its running time is $O(N \log N)$ on two dimensional domains and $O(N^2)$ in higher dimensions where N is the number of triangles in the simplicial mesh. Tarasov and Vyalys [37] extended this algorithm to work in time $O(N \log N)$ on three dimensional domains. Their approach however involved a complicated procedure for dealing with multi-saddle points. Both algorithms suffer from lack of generality and non-trivial treatment of multi-saddle points. Carr et. al [14] introduced an algorithm with running time $O(n \log n + N \alpha(N))$ where n is the number of vertices in the simplicial mesh and α is the notoriously slow growing inverse Ackerman function. This algorithm works in any number of dimensions and has simple treatment of multi-saddle points.

More recent developments in the field focus on extending the existing algorithms to accommodate the distributed [33, 34] and shared memory parallelism paradigms [17, 29]. The focus of this dissertation will be one of the latest developments in a data parallel shared memory algorithm for contour tree computation [17]. Before introducing how that algorithm operates and one of the issues related to its parallel performance we will first give a more detailed overview of the most established serial algorithm [14] on which the data parallel one is based on. In order to talk about any of the two algorithms we must establish some notation and define height graphs and trees as they are defined in [12].

3.3 Height Trees

A height graph is a graph $G = (V, E)$ together with a real valued function h defined on the vertices of G . Height graphs are also known in the literature as weighted graphs. We are changing our notation to be more indicative of the fact that the weight function is defined on the vertices and that it corresponds to height of points in a simplicial mesh. A height tree is a height graph which is a tree. Contour trees are height trees because nodes in the contour tree correspond to nodes in the mesh and can inherit their height (sampled) value. Analogous to the assumption we have made about uniqueness of values we will also assume all vertices in the height trees we consider have unique heights. In other words $h(u) \neq h(v)$ for all $u, v \in V(G)$ where $u \neq v$. The function h naturally induces a total ordering on the vertices. From now on we will assume the vertices of G are given in ascending order. That is to say, $V(G) = \{v_1, v_2, \dots, v_n\}$ where $h(v_1) < h(v_2) < \dots < h(v_n)$. This lets us work with the indices of the vertices without having to compare their heights directly. In this notation $h(v_i) < h(v_j)$ when $i < j$.

Introducing the height function allows us to talk about ascending and descending paths. A

path in a graph is a sequence of vertices (u_1, u_2, \dots, u_k) where $u_i \in V(G)$ for $i \in \{1, 2, \dots, k\}$ and $u_i u_{i+1} \in E(G)$ for $i \in \{1, 2, \dots, k-1\}$. A path in a height graph is ascending whenever $h(u_1) < h(u_2) < \dots < h(u_k)$. If we traverse the path in the opposite direction it would be descending. We will simply call these paths monotone whenever we wish to avoid committing to a specific direction of travel.

When working with height graphs it is useful to extend the definition of a degree of a vertex by taking the height function into account.

Definition 14. *Let G be a height graph and v a vertex of G . The up degree of v is defined as the number of neighbours of v with higher value. It is denoted as $\delta^+(v) = |\{u \in N(v) : h(u) > h(v)\}|$.*

The down degree of a vertex v is defined analogously as $\delta^-(v) = |\{u \in N(v) : h(u) < h(v)\}|$. In the context of height trees the definitions of up and down degrees of a vertex allow us distinguish between two types of leaves - lower and upper leaves.

Definition 15. *Let G be a height graph and v a vertex of G . If $\delta^+(v) = 1$ and $\delta^-(v) = 0$ then v is a lower leaf.*

If $\delta^+(v) = 0$ and $\delta^-(v) = 1$ then v is an upper leaf. We will see in the next section how differentiating between the two types of leaves is a critical part in the computation of the contour tree.

3.4 Serial Algorithm

The contour tree is a tree that consists of [38]:

- Vertices or supernodes that correspond to level sets that contain a critical point.
- Edges or superarcs correspond to path-connected regions bounded by two level sets which both contain a critical point. They connect the supernodes those level sets correspond to.

The contour tree contains information of two types of events - joining and splitting of contours. We can derive two other height trees from the contour tree that each contain the information of the joining and splitting events separately. These are called the join and split trees ???. The join tree contains information for the contours that join together and the split tree holds the information for the contours that split apart. The join tree of a contour tree summarises the evolution of the connectivity of the sublevel sets of the interpolation function and the split tree of the superlevel sets. You can find an example of the join and split trees of Figure 3.2.

The reason we would like to study join and split trees is that the contour tree can be reconstructed from them. The core idea of the algorithm we will present is that we can derive the join and split trees directly from the simplicial mesh and then combine them to obtain the contour tree. We will first describe how the join and split trees are computed from the mesh. We only have to describe the process for the join tree because the computation of the split tree is symmetrical [14].

Definition 16. *A join component is a connected component in the superlevel set $f^{-1}([h, \infty))$ at some $h \in \mathbb{R}$.*

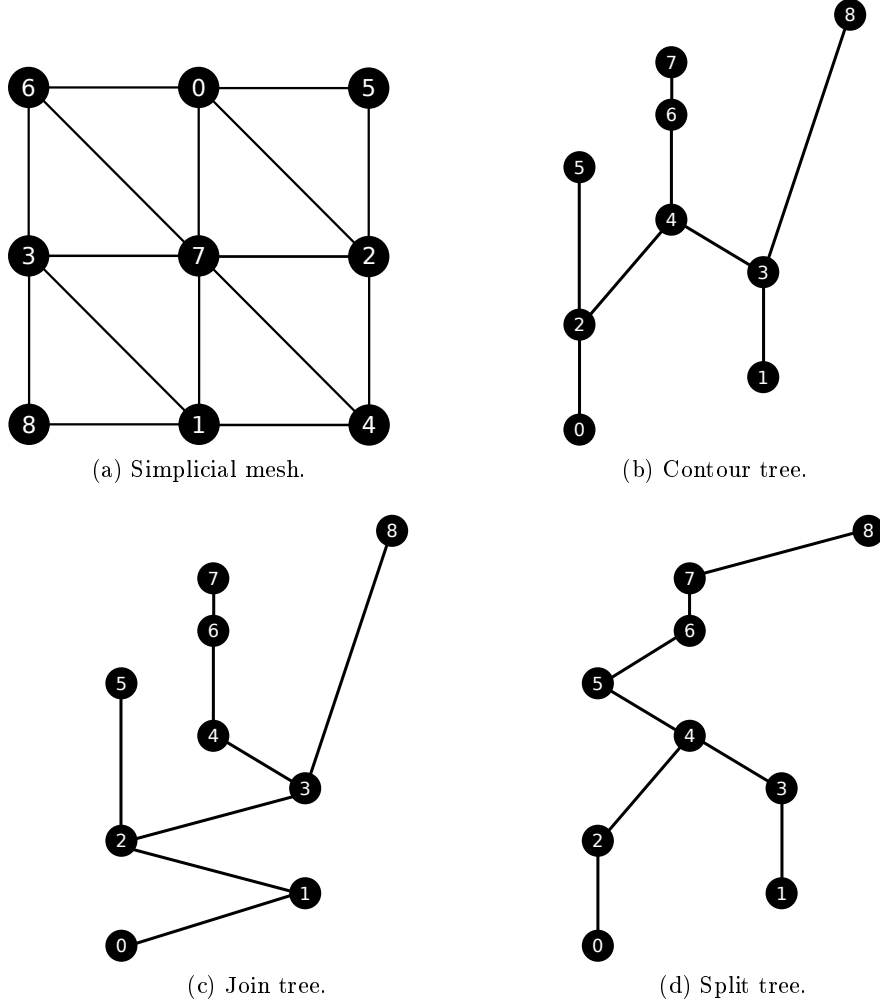


Figure 3.2: The simplicial mesh, join and split trees and contour tree.

Let M be the simplicial mesh from Figure 3.2 (a) and let $h : M \rightarrow \mathbb{R}$ be the interpolation function defined on it. We will refer to h as the height function. To construct the join tree we are going to have to keep track of which components merge together in the superlevel sets of h . We will consider all superlevel sets $M^t = h^{-1}([t, \infty)) = \{x \in M : h(x) \in [t, \infty)\}$ as a one parameter family $\{M^t\}_{t \in \mathbb{R}}$ of nested subsets of M . We can see from this definition that $M^a \subseteq M^b$ whenever $a \leq b$. What the join tree captures is how the connectivity of the superlevel sets changes as the parameter t is increased. The connectivity of superlevel sets changes either at local minima where a new component is created or a saddle point that merges two or more join components.

To visualise this process we can contract every join component to a point much like we did in the Reeb graph. The only difference here is that the equivalence relation is defined for all points in a superlevel set $h^{-1}([t, \infty))$ instead of a level set $h^{-1}(\{t\})$. Because of this change and because join components can only merge the join tree is a tree [22]. Furthermore if $M_m = M$ is the last superlevel set for some $m \in \mathbb{R}$ then all join components merge into one because M is path connected.

We will briefly outline the algorithm for constructing the join tree and refer the reader to [14]

for further implementational details. We know that all critical points are vertices of M and that it is only at the critical points that changes in the topology of the superlevel sets can happen. The algorithm works by considering the vertices of the simplicial mesh in ascending order of their height. If the current vertex is a local minimum we directly add it in the join tree because it starts a join component. If the current vertex is a saddle that joins two or more components (join saddle) we add it to the join tree and add an edge between it and the local minima of the join components it merges. At the end of the computation all vertices will be in the same join component. In order to keep track of which join components different vertices belong to we can use the union-find data structure. The term $\alpha(n)$ in the time complexity of the contour tree algorithm comes from the basic operations of find and search in the union-find data structure.

Not all vertices of the mesh will be in the join tree. Only those which correspond to local maxima and to join saddles. This will pose a problem later on when we wish to combine the join and split trees. To avoid this problem we can augment the join tree by adding all missing vertices. This is done through edge subdivision. Let a and b be two adjacent vertices in the join tree. Let $\{v_1, v_2, \dots, v_n\}$ be vertices in the mesh that are not in the join tree that are given in ascending order in terms of height. Suppose that $h(a) < h(v_i) < h(b)$ for all $i \in \{1, 2, \dots, n\}$ and the vertices v_i are in the same connected component of $X_b - h^{-1}(\{b\}) = h^{-1}((-\infty, b))$. In order to augment the join tree with the first vertex we subdivide the edge ab and label the new vertex as v_1 . Next we subdivide v_1b and label the new vertex as v_2 . We continue to do so and on the k th step we subdivide the edge $v_{k-1}b$ and label the new vertex as v_k .

The procedure of augmentation can be applied to the split tree and contour tree as well. We can use it to augment the contour tree with all vertices of the mesh which are not critical points. This is why we will differentiate between the contour tree and the augmented contour tree. The augmented contour tree contains all regular vertices of the simplicial mesh.

The second step of the algorithm is to combine the join and split trees to produce the contour tree. We will actually combine the augmented join tree with the augmented split tree to obtain the augmented contour tree. Removing the augmentation of the contour tree is left as an optional final step. The first step in merging the two is to identify all leaves of the contour tree and their incident edges. We can recognize them immediately from the join and split trees using the following property [12].

Property 1. *Let v be a vertex such that its up degree in the join tree is 0, its down degree in the split tree is 1 and u is its only down neighbour in the split tree. Then v is an up leaf in the contour tree and vu is an edge in the contour tree.*

There is an analogous property in the case of down leaves and their adjacent edges.

Property 2. *Let v be a vertex such that its up degree in the join tree is 1, its down degree in the split tree is 0 and u is its only down neighbour in the join tree. Then v is a down leaf in the contour tree and vu is an edge in the contour tree.*

Now suppose that we have identified v as a leaf and vu as its adjacent edge in the split or join tree. Another property [12] tells us that if we perform vertex contraction on v (remove v and form a clique from its neighbourhood) from the join, split and contour trees we obtain the join and split trees of the contour tree with v removed. This allows us to iteratively remove

leaves from the join and split trees, add them to the contour tree and then delete them from the join and split tree. We can repeat this process until we have removed all vertices from the join and split trees and all vertices are present in the contour tree. For a detailed description of this process we refer the reader to [14].

The Serial algorithm for the construction of the contour tree is a summary of the results we have obtained so far:

Step 1. Read input data and convert it to a simplicial mesh.

Step 2. Compute the augmented join and split trees from the simplicial mesh.

Step 3. Iteratively remove leaves from the augmented join and split trees and add them to the augmented contour tree until the augmented join and split trees are empty.

Step 4. Remove regular vertices from the augmented contour tree by contracting them.

3.5 Parallel Algorithm

The data parallel contour tree algorithm [17] is largely based on the serial contour tree algorithm we just described. The parallel approach borrows the two phase methodology of computing the join and split trees and then merging them. We will omit describing the process of parallelising join/split tree computation because it is not directly related to the issue we aim to address. We will describe how the merge phase is parallelised in detail.

The data-parallel paradigm works best when there are a large number of computational tasks to be carried out independently. Dependant tasks require some form of synchronisation. Synchronisation is costly in terms of performance. Removing a leaf in the merge phase of the serial algorithm requires little synchronisation with other vertices because it is a local operation. It only involves a few of the vertices of the join and split trees. This means that once we identify all up and down leaves we can remove them in parallel in a single iteration. The key problem to solve in the merge phase is to reduce the number of total iterations needed to remove all vertices from the join and split trees. The amount of parallelism in this computation is limited by the number of leaves at each iteration. For example a tree which is a path of length n will take at least $n/2$ iterations and a tree with one central vertex and n leaves adjacent to it will take only two iterations.

In a graph with no vertices of degree two at least half of the vertices are leaves (see Appendix C). If at every iteration half of the remaining vertices are leaves the total number of iterations would be logarithmic in the number of vertices in the contour tree. In order to ensure this logarithmic collapse Carr et. al [17] have come up with a way of batching some of the paths that start at a leaf and consist of vertices of degree two, in a single iteration. We will call such paths leaf chains. The process of removing them in a single iteration is in effect equivalent to contracting all vertices in the tree of degree two. This leaves only leaves and vertices of degree three or higher and ensures the logarithmic collapse.

The main issue that arises is that leaf chains which are not monotone paths cannot be processed

in a single iteration. They require multiple iterations to process. When some of the vertices in the leaf chains have alternating height and we plot them according to their height they form a characteristic zigzag pattern. We will call paths W-Structures. See for example the path $(5, 2, 4, 3, 8)$ in the contour tree on Figure 3.2. These w-structures are the core issue we are addressing in this dissertation. We would like to obtain a better understanding of them and how and why they affect computation. The first step to solving such a problem is understanding it. The next chapter will address this by developing algorithms that analyse contour trees and determine the largest w-structures that is present in them.

The theoretical issue caused by the w-structures becomes evident in the algorithmic analysis of the parallel contour tree algorithm. According to that the key question in the merge phase of the algorithm is how many iterations are needed to collapse the contour tree. Each iteration takes $O(1)$ steps, because all leaves can be processed in parallel, and $O(t)$ work, where t is the number of leaves. This leads to an overall complexity of $O(\log(t))$ steps and $O(t \log^2(t))$ work if we assume that no w-structures are present. If however there is a w-structure with more zigzags than $\log(t)$ then the authors of the paper claim that the best formal guarantee they can give for the steps is the diameter of the contour tree. One of our goals in analysing the w-structures is to provide a better bound than the diameter of the tree. We will demonstrate how this can be done by developing some new theory about the w-structures in Chapter 4 and through an empirical analysis in Chapter 7.

3.6 Contour Tree Simplification

Finally we will introduce the topic of contour tree simplification. A central problem in using contour trees in visualisation is simplifying their output and presenting only the most important parts to enable human comprehension. The complexity of a contour tree of a large enough data set could severely limit its use. This is why it is vital to employ techniques that simplify the contour trees by removing parts of them that correspond to less "significant" topological features or sampling noise and error. This process helps to reveal the fundamental topological structures present in data.

One technique for contour tree simplification is branch decomposition [35]. Branch decomposition involves decomposing the contour tree into a set of edge-wise disjoint monotone paths (branches) which cover all edges of the tree. The trivial branch decomposition of any tree is obtained by taking every edge to be a separate branch. A branch decomposition is hierarchical when there is exactly one branch that connects two leaves and every other branch connects a leaf to an interior node. An example of a hierarchical branch decomposition is shown in Figure 3.3.

The branches in this scheme represent pairs of critical points. This pairing of critical points forms the basis for a topological simplification. The topological simplification consists of removing branches that do not disconnect the tree. This produces a hierarchy of cancellations like in Figure 3.3. We define the persistence of a branch to be the bigger of the difference between its end points and the persistence of its children. Branches of high persistence reflect more promi-

persistent homology and determine whether the two are equivalent.

Chapter 4

W-structures - Theory and Algorithms

This chapter continues our discussion on the w-structures in a more formal setting. Our contribution in this chapter is to develop new theory that captures their informal description we outlined previously and to use it to construct three algorithms for the detection of the largest w-structure in a height tree. We will describe the algorithms with pseudocode and provide the reader with proofs of their correctness. Finally we will also derive formal bounds on the time and space complexity of the proposed algorithms.

4.1 Formal Description of the w-structures

We are interested in describing paths in height trees which form a characteristic zigzag pattern we described in Chapter 3. Let us first establish some of the basic notation we shall make use of. A path in a graph is a sequence of distinct and adjacent vertices. When dealing with paths in trees we will refer to them through their first and last vertex, because there is a unique path between any two vertices in a tree. For example when dealing with the path v_1, v_2, v_3, v_4 we will denote it with the shorthand $v_1 \rightsquigarrow v_4$. A subpath P' of a path P is a path whose vertices are also vertices of P . We will denote it as $P' \subseteq P$.

The first important property of paths in height trees is their monotone path decomposition. The monotone path decomposition of a path is a sequence of vertexwise maximal monotone subpaths which share exactly one vertex and have alternating direction. An example of the path decomposition of a path is shown in Figure 4.1. If P is a path in a height tree we can decompose it into a sequence of monotone paths P_1, P_2, \dots, P_k such that $P_i \subseteq P$ for $i \in \{1, 2, \dots, k\}$, $|P_i \cap P_{i+1}| = 1$ and $P_i \cup P_{i+1}$ is not a monotone path for $i \in \{1, 2, \dots, k-1\}$ and $k \geq 1$. We can use the number of paths in the monotone path decomposition to characterise paths in height trees. To simplify this characterisation note that the number of subpaths in the monotone path decomposition is one more than the number of vertices where an ascending subpath ends and a descending subpath begins (or vice versa). We shall name those special vertices kinks. In the example path on Figure 4.1 these vertices are 3 and 6.

A kink in a path is a vertex whose two neighbours are either both higher or both lower (Figure 4.2). Given the path (u_1, u_2, \dots, u_k) an inside vertex $u_i \neq u_1, u_k$ is a kink when $h(u_i) \notin (\min(h(u_{i-1}), h(u_{i+1})), \max(h(u_{i-1}), h(u_{i+1})))$. To avoid this cumbersome expression we shall adopt a slight abuse of notation and in the future write it as $h(u_i) \notin$ or $\in (h(u_{i-1}), h(u_{i+1}))$ where it will be understood that the lower bound of the interval is the smaller of the two and the upper bound the larger.

We can use the number of kinks in a path to define a metric on it. We will call this metric the w-length of a path and use it to measure the number of inside vertices of a path which are

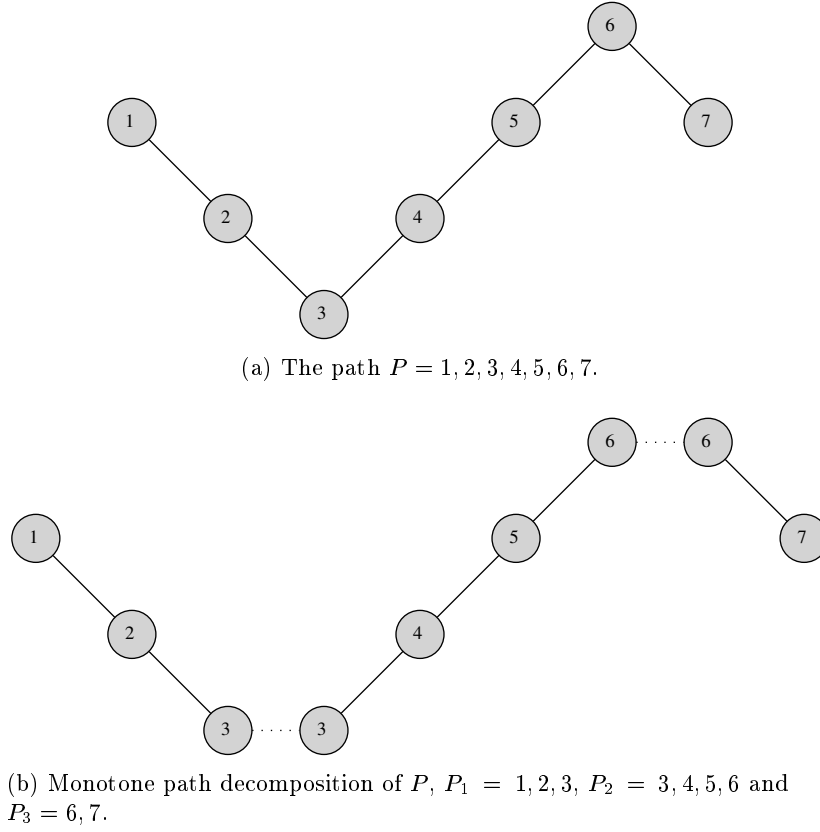


Figure 4.1: A path and its monotone path decomposition.

kinks. This is similar to how the length of a path is a metric that measures the number of edges between its vertices. The notation we will adopt for the w-length and length of a path $u \rightsquigarrow v$ is $w(u, v)$ and $d(u, v)$ respectively. There is no ambiguity here because as we have already said there is a unique path between any two vertices in a tree. One thing we can already claim is that $w(u, v) \leq d(u, v)$ for any two vertices in a height tree. The length of a path with n vertices is $n - 1$, but at only the inside vertices of a path can be kinks. The number of inside vertices in a path of length $n \geq 2$ is $n - 2$.

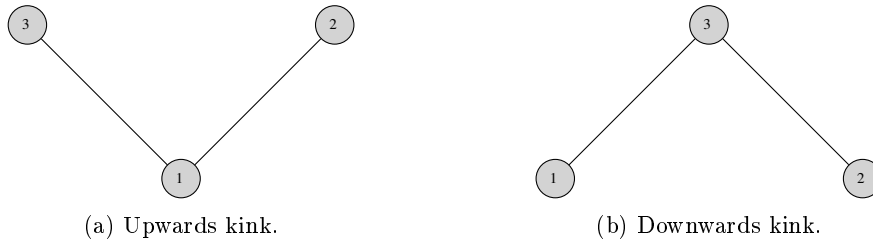


Figure 4.2: Two possible types of kinks (vertices are labeled with their height).

In Chapter 3 we foreshadowed our intention of obtaining the largest w-structure in a contour tree. We can now put this in more precise terms as the path in a height tree that has the maximum w-length (or the longest w-path). We can immediately obtain a brute force approach for this problem by considering all paths in the contour tree and computing their w-length to find the maximum one. This can be expressed with the following optimization term

$$\max_{u,v \in V(T)} \{w(u, v)\}. \quad (4.1)$$

The search space is quadratic in the number of vertices and measuring the w-length of a given path can be done by inspecting the height of every inside vertex and its two neighbours in the path. The worst case time complexity of this algorithm is $O(dn^2)$ where d is the diameter (longest path) of the tree and n is the number of vertices. This is far from satisfactory given that the worst case time complexity of the algorithm for computing the contour tree is close to linear. We can in fact do better.

In Chapter 2 we described two algorithms for obtaining the diameter of a tree. The analogy we made between w-length and length is that both are a metric on paths in a tree. Therefore it may be possible to modify those two algorithms to obtain the longest w-path instead of the longest path. We will call the longest w-path in a tree its w-diameter. Before showing how we can adapt the two tree diameter algorithms we need to establish the two key properties that will play a crucial role in proving the correctness of the two new algorithms.

Definition 17. (*Symmetry Property*) Let $a \rightsquigarrow b$ be a path. Then $w(a, b) = w(b, a)$.

This property is true because the path $a \rightsquigarrow b$ contains the same vertices as the path $b \rightsquigarrow a$.

Definition 18. (*Subpath Property*) Let $a \rightsquigarrow b$ be a path and $c \rightsquigarrow d$ its subpath. Then $w(a, b) \geq w(c, d)$.

This property follows from the fact that all kinks of the path from c to d are also kinks of the path from a to b . An important thing to note is that in the case of path length if one of the paths is a proper subpath of the other then the inequality is strict. This does not have to be the case with w-paths because the w-length decreases only when we reduce the number of kinks in the path.

Definition 19. (*Path Decomposition Property*) Let $a \rightsquigarrow b$ be the path $(a, u_1, u_2, \dots, u_k, b)$ and u_i be an inside vertex for $i \in \{1, 2, \dots, k\}$. Then

$$w(a, b) = w(a, u_i) + w(u_i, b) + w_{a \rightsquigarrow b}(u_i)$$

, where

$$w_{a \rightsquigarrow b}(u_i) = \begin{cases} 0 & \text{if } h(u_i) \in (h(u_{i-1}), h(u_{i+1})) \text{ // } u_i \text{ is not a kink} \\ 1 & \text{otherwise // } u_i \text{ is a kink.} \end{cases}$$

To see why this property is true observe that u_i can be a kink in the path from a to b , but it cannot be a kink in the paths from a to u_i and from u_i to b because it is an endpoint of both. All other possible kinks besides u_i are accounted for by either $w(a, u_i)$ or $w(u_i, b)$. Therefore when making use of path decomposition property we must account for whether the vertex we are decomposing a path at is a kink in that path or not.

4.2 Linear Time Algorithm - 2xBFS

Let us first explore how the Breadth First Search based tree diameter algorithm can be adapted to compute the w-diameter of a height tree. We will call the adaptation 2xBFS for short and it will follow exactly the same steps. The difference is that we will make a modification of the standard BFS algorithm. We will modify it to compute w-distances from a given root vertex to all other vertices in the tree. The algorithm works by first running the modified BFS from any vertex in the height tree and then records the leaf that is farthest in terms of w-length. It then runs the modified BFS a second time from that vertex and again records the farthest vertex from it.

The pseudocode for this algorithm is presented in Algorithm 1. In the algorithm we use two properties of vertices. The property $u.d$ for $u \in V(T)$ is the w-distance of a vertex from the root and the property $u.\pi$ is the parent of a vertex in the modified BFS. We use the function $h(u)$ for the height of a vertex and we use the variable *furthest* to record the furthest vertex from the root in terms of w-length. Note that because of the comparison $u.d \geq \text{furthest}.d$ the furthest vertex will always be a leaf of tree. This comparison forces us to break ties in w-distance with distance. Theoretically it does not have to, but it will simplify some of our notation and proofs. Therefore we will assume that the w-diameter in a height tree is always between two leaves.

Algorithm 1 Computing the W Diameter of a Height Tree.

```

1: function W_BFS(T, root)
2:   root.d = 0
3:   root. $\pi$  = root
4:   furthest = root
5:   Q =  $\emptyset$ 
6:   Enqueue(Q, root)
7:   while Q  $\neq \emptyset$  do
8:     u = Dequeue(Q)
9:     if u.d  $\geq$  furthest.d then
10:      furthest = u
11:     for all v  $\in N(u)$  do
12:       if v. $\pi$  ==  $\emptyset$  then
13:         v. $\pi$  = u
14:         if h(u)  $\notin$  (h(v), h(u. $\pi$ )) then
15:           v.d = u.d + 1
16:         else
17:           v.d = u.d
18:         Enqueue(Q, v)
19:   Return furthest
20: function CALCULATE_W_DIAMETER(T)
21:   s = <any vertex>
22:   u = W_BFS(T, s)
23:   v = W_BFS(T, u)
24:   return v.d

```

This algorithm however is not guaranteed to produce an optimal solution. It may fail to produce the tree's w-diameter, but we can bound the error in terms of the w-diameter. The

correctness of the algorithm is based on the following Lemma.

Lemma 2. *The farthest leaf in terms of w -length from any vertex in a height tree is guaranteed to be the endpoint of a path whose w -length is at least that of the w -diameter minus two.*

Proof. Let T be a height tree and $s \in V(T)$ be the initial vertex we start the first search at. After running the modified BFS twice we obtain two vertices u and v such that:

$$w(s, u) \geq w(s, t), \forall t \in V(T), \quad (4.2)$$

$$w(u, v) \geq w(u, t), \forall t \in V(T). \quad (4.3)$$

Let a and b be two leaves that are the endpoints of a w -path whose length is that w -diameter of T . For any such pair we know that:

$$w(a, b) \geq w(c, d), \forall c, d \in V(T). \quad (4.4)$$

By this equation we have that $w(a, b) \geq w(u, v)$. Our goal in this proof will be to give a formal lower bound on $w(u, v)$ in terms of $w(a, b)$. Let t be the first vertex in the path between a and b that the first BFS starting at s discovers. Note that t cannot be a or b unless s is equal to a or b . The proof will be split into several cases depending on the relative positions of s, t, a, b and u .

Case 1. When the path from a to b does not share any vertices with the path from s to u .

Case 1.1. When the path from u to t goes through s .

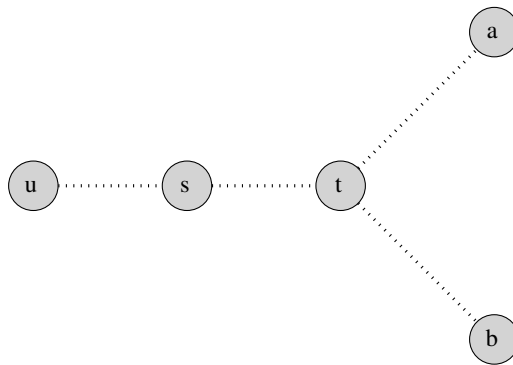


Figure 4.3: Relative position of vertices in Case 1.1 (dotted lines are paths).

In this case $s \rightsquigarrow u$ is a subpath of $t \rightsquigarrow u$. This means that $w(t, u) \geq w(s, u)$. By equation 4.2 we also have that $w(s, u) \geq w(s, a)$. We can therefore conclude that $w(t, u) \geq w(s, a)$. As $t \rightsquigarrow a$ is a subpath of $s \rightsquigarrow a$ then $w(s, a) \geq w(a, t)$. Upon combining these inequalities we obtain that $w(t, u) \geq w(a, t)$.

The vertex t is a inside vertex for both the path $a \rightsquigarrow b$ and the path $u \rightsquigarrow b$. We can use the

path decomposition property for both paths at t as follows:

$$w(a, b) = w(b, t) + w(t, a) + x$$

$$w(u, b) = w(b, t) + w(t, u) + y.$$

where $x = w_{a \rightsquigarrow b}(t)$ and $y = w_{u \rightsquigarrow b}(t)$. In other words $x, y \in \{0, 1\}$ depending on whether there is a kink at t for the path from a to b and from u to b respectively. As $w(t, u) \geq w(a, t)$ we can show that:

$$w(u, b) \geq w(b, t) + w(t, a) + y$$

$$w(u, b) \geq w(b, t) + w(t, a) + x - x + y$$

$$w(u, b) \geq w(a, b) - x + y$$

$$w(u, b) \geq w(a, b) + (y - x)$$

But as $w(u, v) \geq w(u, b)$ (by equation 4.3) we obtain that:

$$w(u, v) \geq w(a, b) + (y - x)$$

Considering all possible values that x and y can take, we can see that the minimum value for the right hand side of the inequality is at $y = 0$ and $x = 1$. This occurs when t is a kink in the path from a to b but not a kink in the path from u to b .

The final conclusion we may draw is that $w(u, v) \geq w(a, b) - 1$.

Case 1.2. When the path from u to t does not go through s .

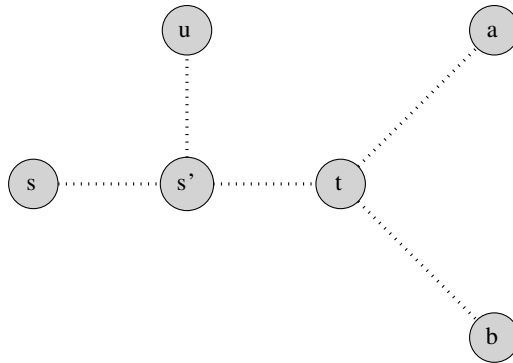


Figure 4.4: Relative position of vertices in Case 1.2 (dotted lines are paths).

If the path from u to t does not go through s then the paths $s \rightsquigarrow t$ and $s \rightsquigarrow u$ have a common subpath. Let s' be the last common vertex in that subpath. We will produce a proof that is similar to the previous case by considering s' in the place of s . We must only account for whether s' is a kink in one of the paths $s \rightsquigarrow u$ or $s \rightsquigarrow a$. Through path decomposition of $s \rightsquigarrow a$ and $s \rightsquigarrow u$ at s' we obtain that:

$$w(s, a) = w(s, s') + w(s', a) + x$$

$$w(s, u) = w(s, s') + w(s', u) + y,$$

where $x = w_{s \rightsquigarrow a}(s')$ and $y = w_{s \rightsquigarrow u}(s')$. By equation 4.2 we know that $w(s, u) \geq w(s, a)$ and therefore:

$$w(s, s') + w(s', u) + y \geq w(s, s') + w(s', a) + x$$

$$w(s', u) + y \geq w(s', a) + x$$

$$w(s', u) \geq w(s', a) + (x - y).$$

We know that $w(t, u) \geq w(s', u)$ because $s' \rightsquigarrow u$ is a subpath of $t \rightsquigarrow u$, so

$$w(t, u) \geq w(s', a) + (x - y).$$

From the fact that $t \rightsquigarrow a$ is a subpath of $s' \rightsquigarrow a$ it follows that $w(s', a) \geq w(t, a)$. This allows us to infer that:

$$w(t, u) \geq w(t, a) + (x - y).$$

Now we are ready to proceed in a similar manner as the previous case. We will decompose the paths from b to a and from b to u at the vertex t as follows:

$$w(b, a) = w(b, t) + w(t, a) + z$$

$$w(b, u) = w(b, t) + w(t, u) + w$$

where $z = w_{b \rightsquigarrow a}(t)$ and $w = w_{b \rightsquigarrow u}(t)$. Then by $w(t, u) \geq w(t, a) + (x - y)$ we have that:

$$w(b, u) \geq w(b, t) + w(t, a) + (x - y) + w$$

$$w(b, u) \geq w(b, t) + w(t, a) + z - z + (x - y) + w$$

$$w(b, u) \geq w(a, b) - z + (x - y) + w$$

$$w(b, u) \geq w(a, b) + (x - y) + (w - z).$$

The minimum value for the right hand side of this equation is at $x, w = 0$ and $y, z = 1$. Using the fact that $w(u, v) \geq w(u, b)$ we finally obtain $w(u, v) \geq w(a, b) - 2$.

Case 2. When the path from a to b shares at least one vertex with the path from s to u .

We can do a path decomposition as follows:

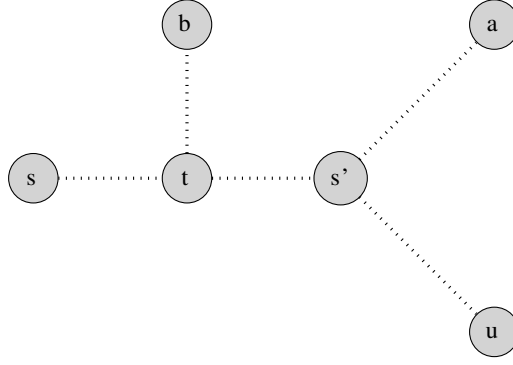


Figure 4.5: Relative position of vertices in Case 2 (dotted lines are paths).

$$w(s, u) = w(s, t) + w(t, u) + x$$

$$w(s, a) = w(s, t) + w(t, a) + y$$

where $x = w_{s \rightsquigarrow u}(t)$ and $y = w_{s \rightsquigarrow a}(t)$. As $w(s, u) \geq w(s, a)$ (by equation 4.2) we obtain that:

$$w(s, t) + w(t, u) + x \geq w(s, t) + w(t, a) + y$$

$$w(t, u) \geq w(t, a) + (y - x)$$

If we again decompose the paths from b to a and from b to u at t we obtain:

$$w(b, a) = w(b, t) + w(t, a) + z$$

$$w(b, u) = w(b, t) + w(t, u) + w$$

where $z = w_{b \rightsquigarrow a}(t)$ and $w = w_{b \rightsquigarrow u}(t)$. Then by $w(t, u) \geq w(t, a) + (x - y)$ we have that:

$$w(b, u) \geq w(b, t) + w(t, a) + (x - y) + w$$

$$w(b, u) \geq (w(b, t) + w(t, a) + z) - z + (x - y) + w$$

$$w(b, u) \geq w(a, b) - z(x - y) + w$$

$$w(b, u) \geq w(a, b) + (x - y) + (w - z).$$

Where similarly to the previous case the rightful conclusion is that $w(u, v) \geq w(a, b) - 2$.

Based on these cases we have shown that that for any input tree the algorithm will produce a w-path that is at most two kinks less than the actual w-diameter of the tree.

□

Let us now show some formal bounds on the time and space complexity of the 2xBFS algorithm.

Lemma 3. *The time complexity of the algorithm is $O(|V|)$.*

Proof. The modified BFS function has the same time complexity as BFS. All we have added to the standard BFS is an "if, then, else" statement. The time complexity of BFS is $O(|V| + |E|)$, but in a tree $|E| = |V| - 1$, so the overall complexity is $O(2|V| - 1) = O(|V|)$. Running the modified BFS function a second time only adds a linear factor the expression and thus the overall complexity of the algorithm is linear. \square

Lemma 4. *The space complexity of the algorithm is $O(|V|)$.*

Proof. The modified BFS function has the same space complexity as the standard BFS. Therefore the space complexity of 2xBFS is $O(|V|)$. \square

4.2.1 Pathological Cases in 2xBFS

Here we will present some examples of pathological cases where the w-diameter outputted by the 2xBFS algorithm differs from the actual w-diameter (Figure 4.6). Each one of the examples corresponds to one of the cases in Lemma 2. In all examples the initial vertex is taken to be s . We have that $w(s, u) = w(s, a) = w(s, b) = 1$, but $d(s, u) > d(s, a) = d(s, b) = 1$. This ensures that u is the last vertex with w-distance 1 visited from the modified BFS. After running the algorithm the vertex outputted by the first BFS function is u . After running the second BFS the longest w-path would be $u \rightsquigarrow a$ or $u \rightsquigarrow b$. We can see that in all figures $w(u, a) = w(u, b) = 1$ or 2, but $w(a, b) = 3$.

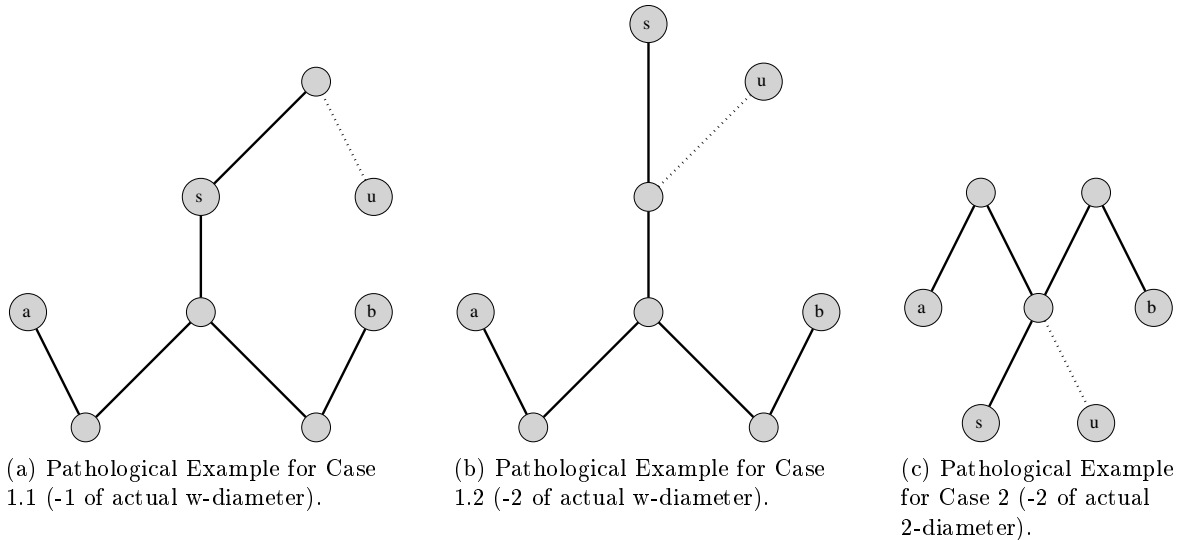


Figure 4.6: Pathological cases in the 2xBFS algorithm (dotted lines are **monotone** paths of length at least three).

If we modified the algorithm to output the closest vertex to the root with maximum w-distance we would obtain the correct output on these example. If however we replace the dotted monotone path with a single edge then u becomes the closest vertex to s and as such will be the one outputted by the first BFS. In that case the algorithm will still output smaller w-path than the w-diameter.

4.2.2 Attempts at resolving the accuracy of 2xBFS

Here we will present two possible ways of improving the accuracy of the output of the 2xBFS algorithm and explain why we were not able to improve upon the theoretical bound of Lemma 2.

One key observation we can make is that on the second run of the BFS we get a w-path that is necessarily longer or equal to one found in the first BFS search. A natural question to ask is whether running the BFS a third, fourth or for that matter n th time would result in the actual w-diameter. On every successive iteration we get a w-path that is longer or equal to the previous one, because w-length is a symmetric path property ($w(a, b) = w(b, a)$). By doing this we can hope that we will eventually obtain a w-path closer to the w-diameter. However there is no guarantee that this will happen. In some cases it is possible that each successive BFS returns the same path over and over again. As an example consider Figure 4.7. If the algorithm starts at s then every successive iterations will go between u and v and then v and u and so on. This is because $w(u, v) > w(u, a) = w(u, b)$ and $w(v, u) = w(v, a) = w(v, b)$, but $d(v, u) > d(v, a) = d(v, b)$.

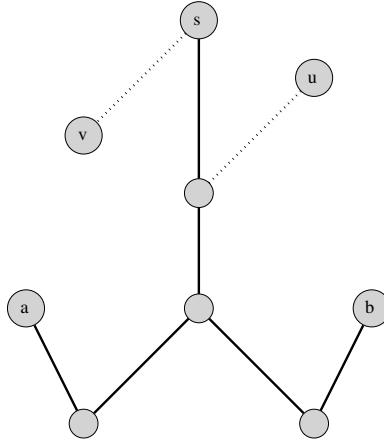


Figure 4.7: Infinite cycle between u and v (dotted lines are monotone paths of length at least 3).

A different heuristic we can apply is to run the algorithm multiple times from different starting vertices and keep the maximum value found. This approach is most reliable when we run the algorithm from all vertices in the tree. Then we will obtain the w-diameter. The issue in doing so is that the time complexity will become quadratic and we would be no better off than with the exhaustive brute force approach. If however we run the algorithm for some subset of the vertices in the tree we lose all guarantees on the accuracy. There may simply be too few vertices from which the algorithm would obtain the actual w-diameter.

4.3 Dynamic Programming Algorithm - DP

It is encouraging that we have obtained an algorithm that bounds the w-diameter but it is also unsatisfactory that we were not able to directly obtain it. To remedy this we will resort to modifying the second tree diameter algorithm. We will use the same optimisation strategy i.e. dynamic programming by making two key changes. Instead of the function $h(u)$ that stores the

height of a subtree with root u we will use the function $w(u)$ that stores the longest w-path that starts at the root of the subtree (the w-height). We will rename the function that stores the value of the optimal solutions for subproblems from $D(u)$ to $W(u)$ accordingly. To summarise $W(u)$ returns the length of the largest w-path in the subtree T_u and $w(u)$ the length of the largest w-path in T_u that starts at u . Note also that in the case of rooted trees we define $N(u)$ to not include the parent of u .

Similarly to the modification of the BFS based algorithm all additional difficulties stem from the difference in the properties of length and w-length of paths. Let us first define the w-height of rooted height tree. It is the longest w-path that starts at the root of the tree. We will now examine how w-height can be computed in manner similar to the height of a rooted tree. Let T be a rooted tree and $s \in V(T)$ be any vertex. Let us also assume the we have computed the w-heights of the children of s . In the case of computing the height we can simply set $h(s) = \max_{u \in N(s)} (h(u)) + 1$. We cannot do so with the w-height because w-length can remain the same if we do not extend the maximum w-path with a kink. To demonstrate this let us assume that $u \in N(s)$ is such that $w(u) = \max_{v \in N(s)} (w(v))$. Then if we wish to extend the maximum w-path that ends at u to s we must account for whether u becomes a kink in it. If none of the children of s with maximum w-height form a kink when extending to s then the w-height of s does not increase.

In order to obtain the w-height of s let u be any of its children and $L_u = \{u_1, u_2, \dots, u_k\}$ be all children of u through which a w-path with length $w(u)$ passes through. We can compute the w-height of s as follows: $w(s) = \max_{u \in N(s)} \{w(u) + \max_{v \in L_u} (w_{s \rightsquigarrow v}(u))\}$. In other words there may be multiple w-paths with the same maximal w-length that end at u . If possible we must pick the one that would make u form a kink with s . If not we can use any of them. It is of no use to consider paths of lesser w-length because when adding s to them the w-length may increase by at most one and match any of maximal paths that go through a vertex in L_u .

The second ingredient in the dynamic programming approach of the tree diameter algorithm was to combine the two longest paths that end at children of the root of the subtree (height paths). As before let T be a tree and s be its root. We first find two distinct children $u, v \in N(s)$ of s such that $h(u)$ and $h(v)$ is maximum amongst all children and $u \neq v$ (otherwise we do not get a proper path). Next we will combine the height paths of u and v in order to obtain the longest path that goes through s . The length of this new path is given by the summation $h(u) + h(v) + 2$. The 2 is added to account for the two additional edges $us, sv \in E(T_s)$. This method of combining paths of course extends to all subtrees in T .

In the case of w-path combinations we must be vigilant of which vertices become kinks in the path combinations. Let us observe a similar scenario where s is the root of a rooted tree height T and $u, v \in V(T_s)$ are two of the children with maximal values for $w(u)$ and $w(v)$. We would ideally like to combine $w(u)$ and $w(v)$ like so: $w(u) + w(v) + w_{u,v}(s)$. This however is not correct. There is a hidden assumption in the sum that the only vertex that can become a kink in this path combination is s . Contrary to this, in fact u and v can also become kinks. Observe that $w(u)$ and $w(v)$ are the w-lengths of two paths. One path starting at u and ending in a leaf of T_u and one starting at v and ending in a leaf of T_v . In the new path both u and v become inside

vertices and depending on whether they become kinks or not the sum may further increase by two. To account for this we must also look at the children of u and v through which a maximum w-path passes. We have already introduced those as L_u and L_v . This process is similar to the one for obtaining the w-height of a vertex and is described by the following formula:

$$\max_{\substack{u,v \in N(s) \\ u \neq v}} \left(h(u) + \max_{t \in L_u} (w_{s \rightsquigarrow t}(u)) + h(v) + \max_{t \in L_v} (w_{s \rightsquigarrow t}(v)) + w_{u \rightsquigarrow v}(s) \right).$$

The longest w-path in a rooted height tree is either entirely contained in one of the subtrees of the root or is a combination of two maximum w-height paths that end at two distinct children of the root. Attention must be paid to one special case. This is when the root of a rooted height tree has exactly one child. We cannot make a path combination in that case so we will use the w-height instead. By combining what we have shown so far we obtain the following expression for the optimal solution:

$$W(s) = \max \begin{cases} \max_{u \in N(s)} (W(u)), \\ \max_{u \in N(s)} \left(h(u) + \max_{t \in L_u} (w_{s \rightsquigarrow t}(u)) \right), \\ \max_{\substack{u,v \in N(s) \\ u \neq v}} \left(h(u) + \max_{t \in L_u} (w_{s \rightsquigarrow t}(u)) + h(v) + \max_{t \in L_v} (w_{s \rightsquigarrow t}(v)) + w_{u \rightsquigarrow v}(s) \right). \end{cases}$$

We will prove the correctness of the algorithm with the following Lemma.

Lemma 5. *The computation for the longest w-path that goes through the root of a subtree is correct.*

Proof. Let T be a rooted height tree and s be its root.

Case 1. s has one child.

When s has exactly one child then the w-length of the longest path that goes through s is equal to the w-height of s by the definition of w-height.

Case 2. s has more than one child.

Let $u', v' \neq s$ be two distinct leaves of T such that the path $u' \rightsquigarrow v'$ is the longest w-path that goes through s . We can decompose the path $u' \rightsquigarrow v'$ at s as: $w(u', v') = w(u', s) + w(v', s) + w_{u' \rightsquigarrow v'}(s)$.

Let u and v be the two children of s through which the path $u' \rightsquigarrow v'$ goes through. The paths $w(u', s)$ and $w(v', s)$ can be further decomposed at u and v respectively. We obtain that $w(u', s) = w(u', u) + w(u, s) + w_{u' \rightsquigarrow s}(u)$ and $w(v', s) = w(v', v) + w(v, s) + w_{v' \rightsquigarrow s}(v)$. In both cases $w(v, s) = 0$ and $w(u, s) = 0$ because u and v are adjacent to s . This means that the paths $u \rightsquigarrow s$ and $v \rightsquigarrow s$ have no inside vertices. Lastly we have that $w_{u' \rightsquigarrow v'}(s) = w_{u \rightsquigarrow v}(s)$ because $u \rightsquigarrow v$ is a subpath of $u' \rightsquigarrow v'$ and both contains s . By substituting these into the first equation we obtain that:

$$w(u', v') = w(u', u) + w_{u' \rightsquigarrow s}(u) + w(v', v) + w_{v' \rightsquigarrow s}(v) + w_{u \rightsquigarrow v}(s).$$

This equation is similar to the expression for the optimal solution. By observing the two carefully we can infer that

$$w(u', u) + w_{u' \rightsquigarrow s}(u) = h(u) + \max_{t \in L_u} (w_{s \rightsquigarrow t}(u))$$

and

$$w(v', v) + w_{v' \rightsquigarrow s}(v) = h(v) + \max_{t \in L_v} (w_{s \rightsquigarrow t}(v)).$$

for otherwise we would be able to assemble a longer w-path that goes through s . This is not possible because we supposed that $u' \rightsquigarrow v'$ is the longest such w-path. Therefore we can conclude that:

$$w(u', v') \leq \max_{\substack{u, v \in N(s) \\ u \neq v}} \{h(u) + \max_{t \in L_u} (w_{s \rightsquigarrow t}(u)) + h(v) + \max_{t \in L_v} (w_{s \rightsquigarrow t}(v)) + w_{u \rightsquigarrow v}(s)\}.$$

The w-path combination on the right hand side of this inequality is valid path in the tree that goes through s . It follows that it cannot be strictly bigger than $w(u', v')$. Therefore they are equal and the computation produces the longest w-path that goes through the root of the tree.

□

Lemma 6. *The DP algorithm produces the w-diameter of a height tree.*

Proof. We just showed the longest w-path through the root of subtree is computed correctly. As the value of the optimal solution is taken in the same way as in the dynamic programming tree diameter algorithm then the correctness of our algorithm follows directly from it.

□

The pseudocode of the proposed w-diameter algorithms is shown in Algorithm 2. The algorithm uses the following arrays:

- $W[u]$ - stores the value of the optimal solution.
- $h[u]$ - stores the w-height.
- $L[u]$ - stores a list of all children through which a w-height path passes.
- $u.\pi$ - stores the parent of a vertex (or itself if is the root).

The algorithm is a modification of the standard DFS algorithm. In the forward phase of the DFS we set the parents of all vertices in order to root tree and sets the optimal solution to all leaves to 0 as they are bases cases. In the backtracking phase of the algorithm was know that all

children of the vertex we are at have been solved for. This is where we implement the recursive formula we developed. The steps of the algorithm are described via comments in the pseudocode.

Next we will show a formal bound on the space complexity of the algorithm.

Lemma 7. *The space complexity of the DP algorithm is $O(|V|)$.*

Proof. The sizes of all arrays we use in the algorithm is linear in the size of the tree. Therefore the space complexity of DP is $O(|V|)$. \square

Let us provide formal bounds on the time and space complexity of the proposed solution. We can summarise the time complexity in the following formula:

$$O\left(|V| + |E| + \sum_{u \in V} \sum_{v \in N(u)} d(v) + \sum_{u \in V} d(u)^2\right),$$

where we use $d(u)$ for the degree of a vertex. The term $|V| + |E|$ comes from executing the Depth First Search, the term $\sum_{u \in V} \sum_{v \in N(u)} d(v)$ is the nested double loop over all children of children of all vertices (on line 15 and 22) and $\sum_{u \in V} d(u)^2$ is the nested double loop over all children in the final path combination (on line 31). We will begin by showing that

$$O\left(\sum_{u \in V} \sum_{v \in N(u)} d(v)\right) = O(|V|).$$

When running DFS on a tree it is not possible to visit a vertex as a child of a child more than once. Suppose for the sake of contradiction that it were possible. Let T be a height tree and suppose we execute DFS to root T from the vertex s . Let u, v be two distinct vertices such that the vertex t is a child of a child of both. Then $s \rightsquigarrow u \rightsquigarrow t \rightsquigarrow v \rightsquigarrow s$ is a cycle in T . Trees have no cycles so this is a contradiction.

Let us now move on to the last term $\sum_{u \in V} d(u)^2$. We can immediately bound it from below via the inequality $\sum_{u \in V} d(u)^2 \geq \sum_{u \in V} d(u) = 2|E|$. This inequality holds because the degree of a vertex is a positive integer and for any $x \in \mathbb{Z}^+$ $x^2 \geq x$. Let us now show how it can be bounded from above.

A triangle is a complete graph on three vertices. Trees have no cycles thus they cannot have induced triangles. Therefore for any edge in a tree $uv \in E(T)$ we have that $d(u) + d(v) \leq |V|$. If there were a vertex t that is in both $N(u)$ and $N(v)$ then $uv, tu, tv \in E(T)$ would be an induced triangle which is a contradiction. If we use this to sum over all edges we obtain that:

$$\sum_{uv \in E(T)} d(u) + d(v) \leq |E||V|.$$

The key to transforming this inequality is to expand the summation $\sum_{uv \in E(T)} d(u) + d(v)$. When it is expanded every term $d(u)$ will be present exactly $d(u)$ times. One time for each one of its adjacent edges and there are exactly $d(u)$ adjacent edges. Therefore:

$$\sum_{u \in V(T)} d(u)^2 = \sum_{uv \in E(T)} d(u) + d(v) \leq |E||V|.$$

To summarise what we have obtained so far:

$$O\left(\sum_{u \in V} \sum_{v \in N(u)} d(v)\right) = O(|V|), \quad O\left(\sum_{u \in V(T)} d(u)^2\right) = O(|V||E|).$$

Therefore a upper bound on the worst case time complexity of our dynamic programming solution is:

$$O(|V| + |E| + |V| + |V||E|) = O(|V||E|).$$

The worst case running time we obtained for the DP algorithm is quadratic. This not good because the time complexity of the brute force approach is quadratic as well. We do however believe that the worst case running time is rarely exhibited and that the algorithm has the potential for good practical performance. One reason that we believe so is that for every vertex of high degree in a tree there are at as many leaves as the degree of that vertex which require constant processing time as the base cases of our recursion. We will however abstain from further theoretical inquiries and instead test this informal hypothesis by implementing both the 2xBFS and DP algorithms and comparing the running time of the implementations. We will do so in Chapter 7.

Algorithm 2 Computing the W Diameter of a Height Tree.

```

1: Function W_DFS(T, s)
2: // Base Case
3: if T.Adj[s] ==  $\emptyset$  AND s. $\pi$   $\neq$  s then
4:   W[s] = 0
5:   h[s] = 0
6:   return
7:
8: // DFS Visit
9: for all u  $\in$  T.Adj[s] do
10:   if u. $\pi$  ==  $\emptyset$  then
11:     u. $\pi$  = s
12:     W_DFS(T, u)
13:
14: // Calculate w-height of s
15: for all u  $\in$  T.Adj[s] do
16:   if L[u] ==  $\emptyset$  then
17:     h[s] = max(h[s], h[u]);
18:   else
19:     for all v  $\in$  L[u] do
20:       h[s] = max(h[s], h[u] + wv,s(u));
21:
22: // Find all children of children that contribute to the a w-height path
23: for all u  $\in$  T.Adj[s] do
24:   if L[u] ==  $\emptyset$  AND h[s] == h[u] then
25:     L[s] = L[s]  $\cup$  u
26:   else
27:     for all v  $\in$  L[u] do
28:       if h[s] = h[u] + wv,s(u) then
29:         L[s] = L[s]  $\cup$  u
30: // Find the maximum w-height path combination
31: maxCombine = 0
32: for all u  $\in$  T.Adj[s] do
33:   for all v  $\in$  T.Adj[s] do
34:     if v == u then
35:       continue
36:     temp = h[u] + h[v]
37:     if L[u]  $\neq \emptyset$  then
38:       for all t  $\in$  L[u] do
39:         if wt,s(u) == 1 then
40:           temp = temp + 1
41:           break
42:     if L[v]  $\neq \emptyset$  then
43:       for all t  $\in$  L[v] do
44:         if wt,s(v) == 1 then
45:           temp = temp + 1
46:           break
47:     if wu,v(s) == 1 then
48:       temp = temp + 1
49:     maxCombine = max(maxCombine, temp)
50: // If there is exactly one child maxCombine will not have been
    maxCombine = max(h[s], maxCombine);

```

Algorithm 3 Computing the W Diameter of a Height Tree. Part 2

```

1: // Find maximum subproblem solution
2: for all  $u \in T.Adj[s]$  do
3:    $W[s] = \max(W[s], W[u])$ 
4: // Take the bigger of the two
5:  $W[s] = \max(W[s], \text{maxCombine})$ 
6: function CALCULATE_W_DIAMETER( $T$ )
7:    $s = \langle \text{any vertex} \rangle$ 
8:    $s.\pi = s$ 
9:    $W\_DFS(T, s)$ 
10:  return  $s.W$ 

```

Chapter 5

Homology

In this chapter we will shift our attention back to algebraic topology and cover the last pieces of background material we need. Those are Homology and Persistent Homology. We will introduce Homology and as tool to analyse the connectivity and number of the holes and voids in a simplicial complex. Using Homology we will define the concept of Persistent Homology. Persistent homology makes use of homology to analyse how the connectivity of a simplicial complex changes as we attach new cells to it.

5.1 Homology

The guiding principle behind the Euler Characteristic was to decompose a space into cells, count them and perform cancellations based on the parity of the dimension of the cells. This approach yields valuable information about a topological space, but we can hope to gain more by generalising it. We shall accomplish this by leveraging the mathematical machinery of Homology. Homology is a tool that was first developed to measure the topological complexity of manifolds [23]. For example with homology we can recognize that there is a hole in the torus and a 3-dimensional volume enclosed in the sphere. The theory of Homology comes in two flavours - **simplicial** and **singular**. Simplicial homology is geared towards analysing simplicial complexes and singular homology its generalisation to arbitrary topological spaces. In this dissertation we will restrict our attention to singular homology because we are primarily interested in the computational aspect of homology.

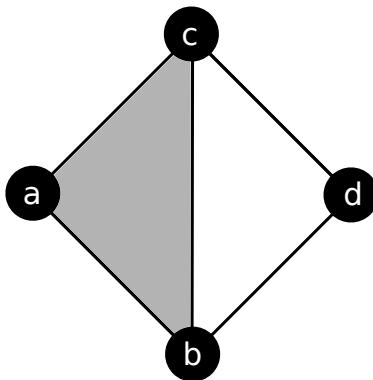


Figure 5.1: A simplicial complex.

Homology is built around the interplay between the two key concepts of **cycles** and **boundaries**. Let us consider the simplicial complex depicted on Figure 5.1 as an example. It consists of four vertices $\{a, b, c, d\}$, five edges $\{ab, bc, ac, bd, bc\}$ and one face $\{abc\}$. Let us first explain what a boundary is. The boundary of a simplex consists of all faces of the simplex whose dimension is one less. We will also call these the codimension-1 faces of the simplex. For example the

boundary of the 1-dimensional simplex (1-simplex) ab consists of the 0-simplices a and b . The boundary of the 2-simplex abc consists of the 1-simplices ab, ac and bc .

A cycle consists of the simplices that form the boundary of a simplex that is of one dimension higher (regardless of whether that simplex is in the complex). In our example we can observe that the edges ab, bc, ac and bd, cd, bc form a 1-dim cycle (1-cycle) because they are the boundary of the 2-simplices abc and bcd . The first simplex abc is in the complex while bcd is not. The definition of one dimensional cycles is similar to the graph theoretic definition of a cycle. The first and last vertex of the paths formed by those edges are the same. A more geometric way explain this is to say that a 1-cycles is a set of edges that completely enclose a 2-dimensional area. To expand this definition to higher dimensional cycles consider the codimension-1 faces of a tetrahedron (four triangles). They form a 2-cycle as they completely enclose a 3-dimensional volume. In general an n -cycle consists of simplices that are the boundary of a $(n+1)$ -simplex.

The interplay between between cycles and boundaries is best described by the following question. Which cycles in the complex are **not** the boundary of a higher dimensional simplex? Such cycles are important because they introduce a void in the complex. Cycles which are the boundary of a higher dimensional simplex can be disregarded because the void they introduce is filled by that higher dimensional simplex. Coming back to our example the cycle ab, bc, ac is the boundary of the 2-simplex abc , but the cycle bd, cd, bc it not the boundary of a simplex that is in the simplicial complex. The cycle bd, cd, bc represents a 2-dimensional hole in the simplicial complex. Note that the cycle ab, bd, cd, ac is in a sense equivalent to the cycle bd, cd, bc because both describe the same 2-dimensional hole in the complex - namely the missing 2-simplex bcd .

Finally observe that the paths formed by the edges bc, ac, ab and ac, ab, bc represent the same cycle. The only difference is which vertex is the starting and ending point. To emphasise that these two cycles are the same we will completely disregard the concept of starting and ending point in a cycle by introducing additive algebraic notation. In this notation the same cycle would be written as $ab + bc + ca$.

Additive notation implies associativity but it does not have the sole purpose of illustrating the point of disregarding edge order. Its more important aspect is that it allows us to treat sums of edges as linear combinations in an abstract vector space. We will work with vector spaces over the field of coefficients $\mathbb{Z}_2 = \{0, 1\}$ together with the standard operations of addition and multiplication modulo two. We will call the elements of these vector spaces n -chains.

Let X be a simplicial complex. An n -chain of X is a formal sum of n -simplices of X [22, p. 79]. The notation we will use for an n -chain is $\sum a_i \sigma_i$ where $a_i \in \mathbb{Z}_2$ and σ_i is an n -simplex of X . We can add two n -chains component wise much like we would add polynomials. For example $(ab + bc) + (ab + cd + bd) = 2ab + bc + cd + bd = bc + cd + bd$ because $2 = 0$ in \mathbb{Z}_2 . Based on the n -chains of X we can define the *chain complex* of X . It is made up of the following vector spaces and linear maps between them:

- The **group of n -chains** $C_n(X)$ of X . These are vector spaces where the vectors are all possible n -chains of X and the coefficients are \mathbb{Z}_2 .
- The **boundary maps** ∂_n between the groups of n -chains of X . These are linear maps

between consecutive groups of n -chains $\partial_n : C_n(X) \rightarrow C_{n-1}(X)$.

What we have defined as the chain complex of X is no more than a collection of vector spaces together with linear maps between. When n is smaller than zero or bigger than the dimension of X then the vector spaces $C_n(X)$ are trivial. They consist only of the zero element. We can visualise the chain complex of X with the so called quiver representation. For our example simplicial complex it would look like:

$$0 \xrightarrow{\partial_3} C_2(X) \xrightarrow{\partial_2} C_1(X) \xrightarrow{\partial_1} C_0(X) \xrightarrow{\partial_0} 0$$

In the general case for an n -dimensional simplicial complex X the full chain complex would be:

$$\dots \longrightarrow 0 \xrightarrow{\partial_{n+1}} C_n(X) \xrightarrow{\partial_n} C_{n-1}(X) \xrightarrow{\partial_{n-1}} \dots \xrightarrow{\partial_1} C_0(X) \xrightarrow{\partial_0} 0 \longrightarrow \dots$$

where we can extend both the right and left-hand sides with the zero vector spaces and zero maps infinitely. More specifically in this sequence ∂_{n+1} and ∂_0 are zero maps. The boundary map ∂_{n+1} sends the zero vector of 0 to the zero vector of $C_n(X)$ and the map ∂_0 sends all vectors in $C_0(X)$ to the zero vector in 0.

Let us now explain how the groups of n -chains and the boundary maps of a simplicial complex are constructed explicitly. In our working example $C_0(X)$ is the vector space that is spanned by all possible linear combinations of the vertices $\{a, b, c, d\}$ of the complex. Therefore $\{a, b, c, d\}$ is a basis for $C_0(X)$ and we write this as $C_0(X) = \text{span}(\{a, b, c, d\})$. A vector in $C_0(X)$ is a linear combination of the basis vectors using coefficients in \mathbb{Z}_2 . Let $\sigma \in C_0(X)$ be an 0-chain, then we can express it as $\sigma = \alpha_0 a + \alpha_1 b + \alpha_2 c + \alpha_3 d$ where $\alpha_i \in \{0, 1\}$ for every $i = 0, 1, 2, 3$.

Going a dimension up $C_1(X) = \text{span}(\{ab, bc, ac, cd, bd\})$. As we pointed out earlier the cycle that consists of the edges bc, cd, bd is represented by the sum or linear combination $bc + cd + bd = 0ab + 1bc + 0ca + 1cd + 1bd$ and has coordinates $(0, 1, 0, 1, 1)$ in $C_1(X)$ with respect to the basis $\{ab, bc, ac, cd, bd\}$. We may of course work use a different basis for the groups of n -chains. For example $C_0(X) = \text{span}(\{a+b, b, c, c+d\})$ because the vectors $(1, 1, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$ and $(0, 0, 1, 1)$ are linearly independent. In this basis the 0-simplex $a + b + c + d$ has coordinates $(1, 0, 0, 1)$.

The boundary maps are defined analogously to how we presented them in the beginning of the section. The effect a boundary map has on a simplex $\sigma \in C_n(X)$ is that it returns the linear combination of the simplices of $C_{n-1}(X)$ that are codimension-1 faces of σ . If σ is the convex combination of the vertices $[v_0, v_1, \dots, v_n]$ then we define its boundary as

$$\partial(\sigma) = \partial([v_0, v_1, \dots, v_n]) = \sum_{i=0}^n [v_0, \dots, \hat{v}_i, \dots, v_n],$$

where the hat on top of v_i in the sum signifies that we omit it in the convex combination. From this definition we can extend ∂ to all n -chains in $C_n(X)$ linearly by allowing it commute with vector addition and scalar multiplication like so:

$$\partial\left(\sum_{\sigma} a_{\sigma}\sigma\right) = \partial\left(\sum_{\sigma} a_{\sigma}[v_{\sigma_0}, v_{\sigma_1}, \dots, v_{\sigma_n}]\right) = \sum_{\sigma} a_{\sigma} \sum_{i=0}^n [v_{\sigma_0}, \dots, \hat{v}_{\sigma_i}, \dots, v_{\sigma_n}].$$

Going back to our working example on Figure 5.1 let $\sigma = ab + bc + ac$ be an n -chain in $C_1(X)$. Then $\partial(ab + bc + ac) = \partial(ab) + \partial(bc) + \partial(ac) = a + b + b + c + a + c = 2a + 2b + 2c = 0$. This examples allows us to observe an important fact. We know that the n -chain $ab + bc + ca$ is a cycle and we obtained that its boundary is zero. This is no coincidence. The defining feature of a cycles is that they have zero boundary. The n -cycles in $C_n(X)$ are defined as the n -chains that go to zero under the boundary map. The set of all vectors that go to zero under a linear map is known as the kernel of the linear map. The kernel of the boundary map $\partial_n : C_n(X) \rightarrow C_{n-1}(X)$ is denoted as:

$$Z_n = \ker(\partial_n) = \left\{ \sigma \in C_n(X) : \partial_n(\sigma) = 0 \right\}.$$

We can also translate the boundaries in the language of linear algebra. The boundaries in $C_n(X)$ are given by the image of $C_{n+1}(X)$ under ∂_{n+1} . We write this as:

$$B_n = \text{im}(\partial_{n+1}) = \left\{ \partial_{n+1}(\sigma) \in C_n(X) : \sigma \in C_{n+1}(X) \right\}.$$

Now that we have the means of describing the cycles and boundaries the only thing that we are missing is a way to partition the cycles into groups that differ from each other only by their boundary. This would make precise the notion that the cycles ab, bd, cd, ac and bd, cd, bc in Figure 5.1 are equivalent because they both represent hole caused by the missing simplex bcd . We must first understand how Z_n and B_n are related through the fundamental Lemma of Homology.

Lemma 8. (*Fundamental Lemma of Homology*) $\partial_{n-1}(\partial_n(\sigma)) = 0$, for every $\sigma \in C_n(X)$.

Proof. We will only sketch the intuitive outline of the proof and refer the reader to [26, p. 105] for a more comprehensive version.

Let us consider the boundary of $\sigma \in C_n(X)$ which is $\partial_n(\sigma)$. It contains all of the $n-1$ faces of σ . Furthermore every $n-2$ face of σ belongs to exactly two $n-1$ faces of σ . Therefore they will cancel out in the second boundary operation $\partial_{n-1}(\partial_n(\sigma)) = 0$. \square

Corrolary 1. For every two consecutive boundary maps ∂_n and ∂_{n-1} in a chain complex $\text{im}(\partial_n) \subseteq \ker(\partial_{n-1})$.

Proof. If the image of ∂_n were not in the kernel of ∂_{n-1} then there would be at least one n -chain σ for which $\partial_{n-1}(\partial_n(\sigma)) \neq 0$. By the Fundamental Lemma of Homology this is not possible. \square

Since $B_n = \text{im}(\partial_{n+1})$ and $Z_n = \ker(\partial_n)$ then B_n is a subset of Z_n . We can make an even stronger statement. From linear algebra [7, p. 59] we know that the kernel and image of a linear map are linear subspaces of the domain and range of the linear map. Therefore B_n and Z_n are linear subspaces of $C_n(X)$. As $B_n \subseteq Z_n$ we can infer that B_n is a linear subspace of Z_n . In order

to partition all cycles in Z_n into equivalence classes of cycles which only differ by a boundary in B_n we can take the quotient of the two spaces. This quotient is what we call the homology of the chain complex.

Definition 20. *The n -th homology group of a chain complex is the quotient $H_n(X) = Z_n/B_n = \ker(\partial_{n+1})/\text{im}(\partial_n)$.*

We know two important things about the quotient $H_n(X)$. The first one is that the quotient of a vector space and its subspace is a vector space [7, p. 91]. The second one is that the dimension of the quotient space is equal to the difference of the dimension of the vector space and the dimension of the subspace [7]. Therefore $H_n(X)$ is a vector space and $\dim(H_n(X)) = \dim(Z_n) - \dim(B_n)$. The elements of $H_n(X)$ are called homology classes. For a cycle $\sigma \in Z_p$ we denote its homology class in the quotient $H_p(X)$ as $[\sigma]$. Two cycles are in the same homology class exactly when they only differ by a boundary. In our working example this means that $[ab + bd + cd + ac] = [bd + cd + bc]$. To verify this let us compute the difference of the two $(ab + bd + cd + ac) - (bd + cd + bc) = (ab + bd + cd + ac) + (bd + cd + bc) = ab + ac + bc$. The n -chain $ab + ac + bc$ is the boundary of the simplex abc . We have shown that both cycles differ by a boundary. Therefore they are both representatives of the same homology class. In the language of abstract algebra we would say that they are in the same coset.

The dimensions of the homology groups are a summary of the topological information about the connectivity of the n -dimensional simplices of a complex. They are called Betti numbers and they have the following interpretation.

- Betti zero or $\beta_0 = \dim(H_0)$ is the number of connected components in a simplicial complex.
- Betti one or $\beta_1 = \dim(H_1)$ is the number one dimensional holes in a simplicial complex or simply holes.
- Betti two or $\beta_2 = \dim(H_2)$ is the number two dimensional holes in a simplicial complex or simply voids.

The higher Betti numbers represent the number of higher dimensional voids. In a simplicial complex of finite dimension the Betti numbers higher than the dimension of the complex are all zero. We refer the reader for examples of more homology computation to [22, p. 88].

5.2 Reduced and Relative Homology

There are two extensions of homology we need to discuss so that we may be able to fully harness the power of persistent homology in the following section. Those are reduced and relative homology.

The need for reduced homology arises from a slight inconsistency in the interpretation of the homology groups. Take for example the simplicial complex that consists of a single vertex. All of its homology groups except for H_0 are trivial. It is convenient in many application to force H_0 to behave like the rest of the homology group. More specifically, we would like for the 0th homology group it to be trivial as well. In order to do this we must force all path-connected simplicial complexes will have a trivial 0th homology group.

The geometrical interpretation of this extension is the reduced 0th homology classes represent the number of voids that completely separate path connected components and not the path connected components themselves [22, p. 83]. In order to accomplish this we will augment the chain complex of a simplicial complex X with one additional group \mathbb{Z}_2 and one linear map $\epsilon : C_0(X) \rightarrow \mathbb{Z}_2$. The resulting chain complex is

$$\dots \longrightarrow C_1(X) \longrightarrow C_0(X) \xrightarrow{\epsilon} \mathbb{Z}_2 \longrightarrow 0 \longrightarrow \dots,$$

where the function $\epsilon : C_0(X) \rightarrow \mathbb{Z}_2$ is defined as $\epsilon(\sum_i n_i \sigma_i) = \sum_i n_i$. The value of ϵ is equal to the parity of the number of simplices in the chain. We will define the reduced homology as the homology of the augmented chain complex or $\tilde{H}_n(X)$. From [26, p. 110] we have that $\tilde{H}_n(X) = H_n(X)$ for $n > 0$ and $\tilde{H}_0(X) \oplus \mathbb{Z}_2 = H_0(X)$. The reduced homology of a chain complex effectively reduces the dimension of the 0th homology group by one.

Another crucial concept is that of relative homology. Relative homology aims to simplify the homology of a simplicial complex X by discarding all chains that belong to a subcomplex A of X . We do so by taking the quotient of the chain groups of X and the chain groups of A . We will define this quotient as $C_n(X, A) = C_n(X)/C_n(A)$ and call $C_n(X, A)$ the relative chain groups. As the boundary maps take $C_n(A)$ to $C_{n-1}(A)$ they induce relative boundary maps from $C_n(X, A)$ to $C_{n-1}(X, A)$. The relative boundary maps take a relative class from $[\sigma] \in C_n(X, A)$ to a relative class $[\partial_n(\sigma)] \in C_{n-1}(X, A)$. By taking the relative chain groups together with the relative chain maps we obtain the relative chain complex.

$$\dots \longrightarrow C_n(X, A) \longrightarrow \dots \longrightarrow C_1(X, A) \longrightarrow C_0(X, A) \longrightarrow 0.$$

We will define the relative homology groups of the relative chain complex as $H_n(X, A) = \ker(\partial'_n)/\text{im}(\partial'_{n-1})$ where ∂'_n and ∂'_{n-1} are the relative boundary maps. The most important thing to note is that $H_n(X, A)$ is not the quotient $H_n(X)/H_n(A)$, but the homology of the relative chain complex.

Intuitively here is how we can think of the relative homology classes [26, p. 115].

- A relative chain α is a relative cycle when its boundary $\partial_n(\alpha)$ is in $C_{n-1}(A)$.
- A relative cycle α is trivial in the homology when it is the sum of a boundary $\partial_{n+1}(\beta)$ of $\beta \in C_{n+1}(X)$ and a chain $\gamma \in C_n(A)$.

There is a connection between the relative chain complex and the reduced chain complex [25, p. 69]. In fact they are equal when we quotient by a single vertex of X . Let p be a 0-simplex of X then $\tilde{H}_n(X) = H_n(X, p)$. The reason for this is that the 0th homology class of p becomes trivial when we quotient it out.

The relative homology classes are a purely algebraic construction, but for simplicial complexes there is an appropriate geometric intuition that goes along with them. It is expressed through the following theorem [22, p. 92].

Theorem 1. (*Excision Theorem*) Let $K_0 \subseteq K$ and $L_0 \subseteq L$ be two pairs of simplicial complexes that satisfy $L \subseteq K$ and $L - L_0 = K - K_0$. Then they have isomorphic relative homology groups

$$H_n(K, K_0) \simeq H_n(L, L_0).$$

A corollary of the Excision Theorem [25, p. 71] is that if A is a subcomplex of X then $H_n(X, A) \simeq H_n(X/A, A/A) \simeq \tilde{H}_n(X/A)$ where A/A is a single point in X/A . This will allow us to leverage our geometric intuition about quotient spaces to compute homology groups. The quotient space of a simplicial complex however is generally not a simplicial complex. Simplicial Homology is well defined for such spaces (known as Δ complexes), but we will not be able to cover this extension in detail.

In this dissertation we will only need to compute the 0th homology of quotient spaces and for this we can use our geometric intuition to visualise them and count the number of connected components. We will make use of this geometric intuition when we are presenting examples in the next chapter. In particular when X is a small enough simplicial complex we will use the Excision Theorem to compute the dimension of $\tilde{H}_0(X/A)$ by simply counting the number of connected components of X/A and subtracting one.

5.3 Inclusion Maps and Induced Maps on Homology

We will devote this section to introducing inclusion maps between chain complexes and how they induce linear maps between the homology and relative homology groups of the chain complexes. We will begin by defining inclusion maps:

Definition 21. *Let X be a simplicial complex and A be a subcomplex of X . A function $i : A \rightarrow X$ is an inclusion map when i takes a simplex σ in A to σ in X .*

In other words $i(\sigma) = \sigma$ and when $A = X$ then the inclusion map is the identity map. Inclusion maps are a special case of a simplicial map [27, p. 52]. A simplicial map between two simplicial complexes takes simplices from one to simplices of the other. We are opting for introducing the special case directly because we will only use inclusion maps in the following sections.

Inclusion maps between simplicial complexes allow us to obtain inclusion maps between the groups of n -chains of A and X .

Definition 22. *Let X be a simplicial complex and A be a subcomplex of X and $i : A \rightarrow X$ be an inclusion map. Then i induces an inclusion map $i_\# : C_n(A) \rightarrow C_n(X)$ for all $n \in \mathbb{Z}$.*

In order to define $i_\#$ we just have to extend i linearly from simplices to n -chains of $C_n(A)$ as follows: $i_\#(\sum a_\sigma \sigma) = \sum a_\sigma i(\sigma)$. Note that $i_\#$ is also an inclusion map because every n -chain in $C_n(A)$ is also an n -chain of $C_n(X)$ and i maps simplices to themselves. Upon obtaining inclusion maps between the chain complexes of A and X we can take a step further and use that inclusion map to induce a linear map between the homology groups of A and X .

Definition 23. *Let X be a simplicial complex and A be a subcomplex of X and $i_\# : C_n(A) \rightarrow C_n(X)$ be an inclusion map. Then $i_\#$ induces a linear map $i_* : H_n(X) \rightarrow H_n(Y)$ such that $i_*([\sigma]) = [i_\#(\sigma)]$ for all $n \in \mathbb{Z}$.*

Where $[\sigma]$ is the homology class of the n -chain σ in $C_n(A)$ and $[i_\#(\sigma)]$ is the homology class of the n -chain $i_\#(\sigma) = \sigma$ in $C_n(X)$. A crucial thing to note here is that i_* does not have to be

an inclusion map between the homology groups. Take for example a cycle which is not trivial in $H_n(A)$. The same cycle could become trivial in $H_n(X)$ if X contain a simplex whose boundary is σ , but A does not not.

Lastly we will expand the definitions we have made so far to relative homology groups.

Definition 24. *Let X be a simplicial complex. Let A and B be two subcomplexes X such that $A \subseteq B$. Then the identity map $i : X \rightarrow X$ induces a linear map $i_* : H_n(X, A) \rightarrow H_n(X, B)$ for all $n \in \mathbb{Z}$.*

The general case of this definition is for two topological spaces X and Y , two subspaces $A \subseteq X$ and $B \subseteq Y$ and a continuous function $f : X \rightarrow Y$ such that $f(A) \subseteq B$. For a demonstration of how the map between $H_n(X, A) \rightarrow H_n(Y, B)$ is induced we refer to [26, p. 118]. The result shown here holds because inclusion maps are continuous and because as $A \subseteq B$ then $i(A) \subseteq i(B)$. We will not have to construct such a maps explicitly we just need the reader to be aware of how they are defined.

5.4 Persistent Homology

We will now turn our attention to one of the tools that has made topological data analysis so viable in recent years. This tool is called Persistent Homology (PH) and it is primarily used for measuring the significance of topological features [40]. The primary motivation for introducing persistent homology in the first place was the practical need to cope with noise in data [21]. The general idea is that once we extract the topological features from data we can attribute a metric to them. We can then use that metric to extract the important topological features by ignoring all features we deem of low significance (they are considered noise).

Persistent homology emerged in the early 2000s in the works of Herbert et. at [23] as a tool for automated topological simplification. The building blocks of persistent homology are sequences of nested simplicial complexes called filtrations. A filtration of a simplicial complex X is a one parameter family of simplicial complexes $\{X_t\}_{t \in \{0, \dots, n\}}$ where $X_i \subseteq X_j$ whenever $i \leq j$ and $X = X_n$. If we arrange the consecutive X_i in a linear sequence we obtain the following:

$$X_0 \subseteq X_1 \subseteq \dots \subseteq X_{n-1} \subseteq X_n = X.$$

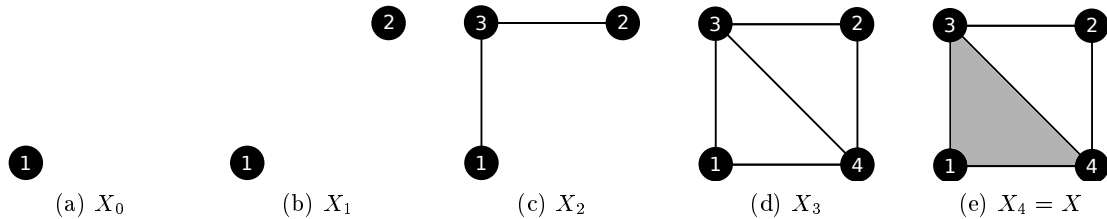


Figure 5.2: Example of a filtration of a simplicial complex.

Another way to think about a filtration is that we start with simplicial complex and iteratively add new simplicies to it. It is customary to call the index of this filtration time to make it more

indicative of a process that evolves over time. The key insight in persistent homology comes from realising that we can track individual homology classes in the homology groups of the X_k as we go from one simplicial complex to the next. This is made possible by the subset relation between subsequent complexes in the filtration. Since X_k is a subcomplex of X_{k+1} we have inclusion maps $i : X_k \rightarrow X_{k+1}$ for $k \in \{0, 1, \dots, n-1\}$. Using those inclusion maps we can build the following chain of simplicial complexes.

$$X_0 \xrightarrow{i} X_1 \xrightarrow{i} \dots \xrightarrow{i} X_{n-1} \xrightarrow{i} X_n.$$

We have already shown in the previous chapter that inclusion maps induce linear maps between homology groups. By invoking this property we obtain the following sequence of homology groups and maps between them:

$$H_p(X_0) \xrightarrow{i_*} H_p(X_1) \xrightarrow{i_*} \dots \xrightarrow{i_*} H_p(X_{n-1}) \xrightarrow{i_*} H_p(X_n)$$

for all $p \in \{0, 1, 2, \dots\}$. The induced linear maps encode the information about the topological changes in the homology of consecutive complexes in the filtration. We will introduce the following terminology to help us interpret this information [25, p. 104]:

- A homology class is **born** if it is not in the image of the homology group of the previous complex in the filtration under i_* .
- A homology class **dies** if its image under i_* is the zero element or it merges with an older class.
- A homology class **persists** if its image under i_* is not the zero element and it does not merge with an older class.

Let us now expand on the case when two classes merge in a filtration. Suppose that $[\alpha] \neq [\beta]$ are two homology classes of some $H_p(X_i)$ such that $i_*([\alpha]) = i_*([\beta])$ in $H_p(X_{i+1})$ because of the introduction of a new boundary. For example the classes $[0]$ and $[1]$ in $H_0(X_1)$ are not equivalent. They represent two distinct connected components. By introducing the edge 01 in X_1 they become one connected component and $[0] = [1]$ in $H_0(X_2)$. We must choose which one of the classes dies and which one of them persists. It does not matter what we choose as long as we commit to the choice in the future.

In order to be consistent in choosing we will apply the Elder Rule [22, p. 150]. According to the elder rule the class whose birth time is smaller will persist and the other one will die. In the case of merging multiple classes all will die except for the oldest. In the example where $[0] = [1]$ in $H_0(X_2)$ we will pick $[0]$ as the representative of the homology class because it is older and say that $[1]$ dies at time 2 because it merges with $[0]$.

Using the language of birth and death we can define the persistence of a homology class. Let α be a homology class that is born in X_i and dies in X_j . We call the difference $j-i$ the persistence of α . Some classes however do not have a defined death time. These are the classes of the final complex in the filtration. We will call those classes essential and set their persistence to ∞ . Classes that have persisted for a large number of timesteps are deemed significant. Ephemeral

classes on the other hand are not. Such classes are often considered to correspond to statistical noise or sampling error.

One way to visualise persistent homology is by producing the so called persistence pairs and plotting them. A persistence pair (t_1, t_2) of a homology class α is a pairing of two timestamps - the birth and death time of α . The essential classes are the exception to this rule. A persistence pair of an essential class is formed as (t, ∞) where their birth time is t and we set their death time to ∞ due to the lack of one.

Persistence pairs are visualised by plotting them as points in the plane. This is called a persistence diagram. We denote the essential classes as triangles above the last simplex in the filtration and other classes as circles (Figure 5.3 a, b). We can see that there are two pairs in the 0th persistent homology of the filtration on Figure 5.2. The first pair is $(1, 2)$ and it corresponds to the birth of the second connected component in time 1 and its death in time 2 where it merges with the first connected component. The second pair $(0, \infty)$ is an essential class that corresponds to the single connected component of X_0 that persists until the end of the filtration at X_4 .

In the 1st persistent homology we have two pairs. One is $(3, 4)$ and the other is $(3, \infty)$. These correspond to two 1-cycles that are born at time 3. Those are $13 + 14 + 34$ and $23 + 24 + 34$. At time 4 the cycle $13 + 14 + 34$ becomes trivial in $H_1(X_4)$ because we add the boundary 234 . The cycle $23 + 24 + 34$ however persists in $H_1(X_4)$ and therefore corresponds to the pair $(3, \infty)$.

Another way to visualize persistent homology is via a barcode diagram [24]. A barcode diagram is a graphical representation of the persistent homology as a collection of horizontal line segments in the plane. The horizontal axis corresponds to the current homology group and the vertical axis corresponds to an arbitrarily chosen basis for the current homology group. A line is drawn between two basis elements $[\alpha] \in H_k(X_t)$ and $[\beta] \in H_k(X_{t+1})$ whenever $i_*[\alpha] = [\beta]$. A line is cut short when a basis element in $H_k(X_t)$ dies upon entering $H_k(X_{t+1})$. The persistent pairs correspond uniquely to lines in the barcode diagram. If there is a persistence pair (t_1, t_2) then we draw a line from t_1 and cut it off at t_2 . For example the barcode diagram for the persistent homology of the filtration on Figure 5.2 is depicted on Figure 5.3 c).

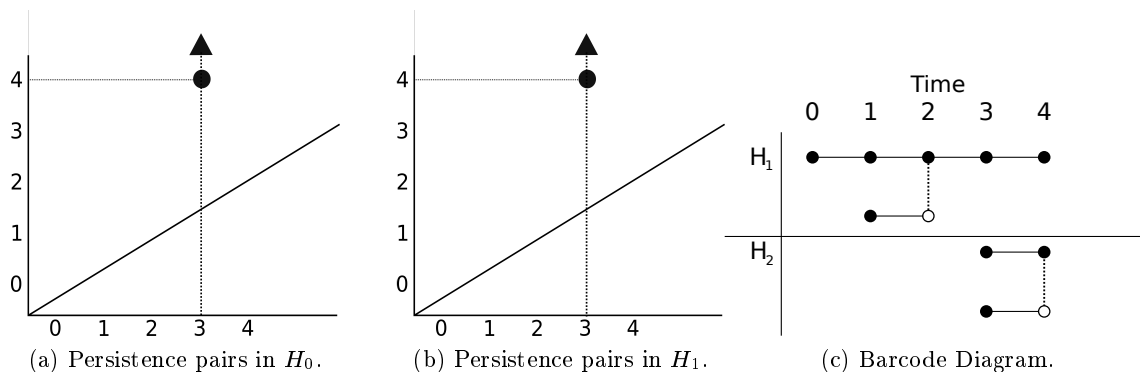


Figure 5.3: Examples of visualising persistent homology of the filtration on Figure 5.2.

Now let us apply the general theory of persistent homology to a more familiar domain. Let M be a simplicial mesh that is the triangulation of a bounded area in \mathbb{R}^2 and let $f : M \rightarrow \mathbb{R}$ be a linear interpolat based on the vertices of M . We would like to obtain a filtration of M that

we can analyse with persistent homology. The filtration that is proposed in the literature [22, p. 161] for manifolds and Morse functions is defined via the sublevel or superlevel sets of the manifold.

From Chapter 3 we know that M is a manifold and that f is a Morse function. The following claims hold:

- f has finitely many critical points,
- changes in the topology of the sublevel sets of M happen only at critical points and
- critical points are at the vertices of M .

Therefore we need only consider the sublevel sets at the critical values of f . Let $c_1 < c_2 < \dots < c_k$ be all critical values and let $M_{c_i} = f^{-1}((-\infty, c_i])$ be the sublevel sets at the critical values where we do not include all simplices which are not entirely contained in M_{c_i} in order to obtain a simplicial complex. This lets us obtain a filtration of the sublevel sets:

$$M_{c_1} \subseteq M_{c_2} \subseteq \dots \subseteq M_{c_{n-1}} \subseteq M_{c_k} = M.$$

From this filtration we can produce the following sequence of homology groups and maps between them:

$$H_n(M_{c_1}) \xrightarrow{i_*} H_n(M_{c_2}) \xrightarrow{i_*} \dots \xrightarrow{i_*} H_n(M_{c_{n-1}}) \xrightarrow{i_*} H_n(M_{c_k}) = H_n(M).$$

If we had taken the superlevel sets of M we would have obtained a different filtration. We will call that the descending filtration of M .

$$H_n(M^{c_1}) \xrightarrow{i_*} H_n(M^{c_2}) \xrightarrow{i_*} \dots \xrightarrow{i_*} H_n(M^{c_{n-1}}) \xrightarrow{i_*} H_n(M^{c_k}) = H_n(M).$$

We can use both filtration to compute the persistent homology of the ascending and descending filtrations of M .

5.5 Extended Persistence

When M is a triangulation of a bounded area in \mathbb{R}^2 and $f : M \rightarrow \mathbb{R}$ is a linear interpolat not all critical points are paired by persistent homology. Some are left out because pairs that include essential homology classes have infinite persistence. Such pairs only pair one critical point, not two. The purpose of extending persistence is to devise a way to pair the critical points that correspond to essential homology classes with remaining unpaired critical points [18].

The main idea behind extended persistence is to follow the ascending filtration of persistent homology with a descending filtration. In the descending filtration we remember which the essential homology classes are. While in the descending filtration once we reach a class that is homologous to an essential class we consider it to be destroyed and thus paired. Extended

persistence consists of two consecutive sequences. The first sequence is the made up of homology groups going up

$$0 \rightarrow H_n(M_{c_1}) \rightarrow H_n(M_{c_2}) \rightarrow \dots \rightarrow H_n(M_{c_{k-1}}) \rightarrow H_n(M_{c_k}) = H_n(M).$$

Just like in ordinary persistence the linear maps between consecutive homology groups are induced by the inclusion maps between $M_{c_i} \subseteq M_{c_{i+1}}$. The second sequence is made up of relative homology groups that come back down:

$$H_n(M) = H_n(M, M^{c_k}) \rightarrow H_n(M, M^{c_{k-1}}) \rightarrow \dots \rightarrow H_n(M, M^{c_2}) \rightarrow H_n(M, M^{c_1}) \rightarrow 0.$$

The linear maps between the relative homology groups in the relative sequence are induced by inclusion of pairs. The pairs (M, M^{c_i}) and $(M, M^{c_{i-1}})$ are such that $M^{c_i} \subseteq M^{c_{i-1}}$ and by Definition 24 induce linear maps between $H_n(M, M^{c_i})$ and $H_n(M, M^{c_{i-1}})$. When we combine the two sequences at $H_n(M_{c_k}) = H_n(M) = H_n(M, M^{c_k})$ we obtain the following single sequence:

$$0 \rightarrow H_n(M_{c_1}) \rightarrow \dots \rightarrow H_n(M_{c_k}) = H_n(M, M^{c_k}) \rightarrow \dots \rightarrow H_n(M, M^{c_1}) \rightarrow 0.$$

The extended persistence sequence of homology groups start from the trivial group and ends at the trivial group. This means that all classes that are born will eventually die in a finite amount of time steps. Extended persistent is symmetric. If we instead start with a descending filtration

$$0 \rightarrow H_n(M^{c_k}) \rightarrow H_n(M^{c_{k-1}}) \rightarrow \dots \rightarrow H_n(M^{c_2}) \rightarrow H_n(M^{c_1}) = H_n(M)$$

and follow that with a relative ascending filtration

$$H_n(M) = H_n(M, M_{c_1}) \rightarrow H_n(M, M_{c_2}) \rightarrow \dots \rightarrow H_n(M, M_{c_{k-1}}) \rightarrow H_n(M, M_{c_k}) \rightarrow 0$$

we would get the extended persistence of the descending filtration of M

$$0 \rightarrow H_n(M^{c_k}) \rightarrow \dots \rightarrow H_n(M^{c_1}) = H_n(M, M_{c_1}) \rightarrow \dots \rightarrow H_n(M, M_{c_k}) \rightarrow 0.$$

A good aid in understanding extended persistence is the Excision Theorem. We can use it compute the relative homology groups $H_n(M, M^{c_k})$ by computing the reduced homology groups of the quotient space $\tilde{H}_n(M/M^{c_k})$. We refer the reader [39] for more details examples of this.

Chapter 6

Extended Persistence and Branch Decomposition

We have seen in the previous chapter that the idea behind persistent homology is akin to that of contour tree simplification. Both are used to measure the topological significance of features in data, but persistent homology is a more general framework. The contribution we will make in this chapter is to explore the relation between the pairs of critical points obtained via branch decomposition and via extended persistence. We will introduce the historical relation between the two in the literature and then proceed to compare them by computing both on an example data set.

6.1 Persistence of Branches

In 2004 Pascucci et. al published the original paper that introduced contour tree simplification through branch decomposition [35]. In that paper it is clearly stated that there is a similarity between the new method they have developed and the more general and recently established method of persistent homology (as it was originally introduced in [23]). They do not go further in describing in detail how the persistence of branches is derived formally within the framework of persistent homology. The relation between the two is not fully explored in the paper nor in subsequent publications. Describing it formally in a complete and rigorous manner is beyond the scope of this dissertation. We will however pose and answer one question that relates branch decomposition and persistent homology. Are the pairs of critical points produced by branch decomposition the same as the pairs produced by the persistent homology of the zeroth homology?

We can immediately show that they are not based solely on the fact that essential classes have infinite persistence and all branches have finite persistence. As we have seen this issue can be remedied by the use of extended persistence, but extended persistence was introduced in a paper by Herbert et. al [18] published five years later in 2009. However the concept of pairing essential classes was probably not unheard of at the time of publishing of [35]. A previous paper [6] by Herbert et. al first published in 2004 outlines a method of pairing the essential homology classes of 2-manifolds. It is completely reasonable to assume that Pascucci et. al were aware of this recent development. Since the method introduced in [6] is a special case of what was later defined as extended persistence we will extend our comparison to the pairs of critical points obtained by extended persistence. Are the pairs of critical points produced by branch decomposition the same as the pairs produced by the extended persistent of the zeroth homology?

6.2 Persistence Pairs vs Branch Decomposition Pairs

In this section we will present an example which demonstrates that at least one pair of critical points is different in branch decomposition and in extended persistence. This example will be based on the simplicial mesh we presented in Figure 3.1 and our familiar w-structures. A defining feature of this example is that the w-structure in the contour tree of the simplicial mesh prevents the existence of a monotone path between the global minimum and global maximum. Since monotone paths in the contour tree correspond to monotone paths in the simplicial mesh [14] then branches obtained via branch decomposition correspond to valid topological cancellations in the simplicial mesh [35]. Therefore the critical points obtained via branch decomposition will correspond to critical points in the simplicial mesh and as such will be directly comparable to critical points produced by the extended persistence of a filtration of the mesh.

Conversely the 0th persistence homology can be computed either on a filtration of the simplicial mesh or on a filtration of the contour tree directly. The output will be the same because the contour tree records the changes in connectivity of level sets of the simplicial mesh. For completeness we will present the ascending and descending filtration of the contour tree in Appendix A and Appendix B, but we will not use that directly here. To conclude we will compute and compare the following:

- The join tree (Figure 6.1 a), split tree (Figure 6.2 a) and contour tree (Figure 6.3 a).
- The branch decomposition of the join tree (Figure 6.1 b), split tree (Figure 6.2 b) and contour tree (Figure 6.3 b).
- Ascending filtration (Figure 6.4) and Descending filtration (Figure 6.5) of the simplicial mesh.
- The extended persistence in the 0th homology of the ascending (Figure 6.6 a) and descending (Figure 6.6 b) filtration of the simplicial mesh.

We will compute the branch decomposition of the join, split and contour trees first. We already computed the branch decomposition of the contour tree in Chapter 3. For the join tree the candidate branches are $(2, 5)$, $(3, 7)$ and $(3, 8)$. We remove them in order of persistence. First we remove $(2, 5)$, then $(3, 7)$ and afterwards we remove the root branch $(0, 8)$ as only remaining branch. In the split tree there are only two candidate branches. Those are $(0, 4)$ and $(1, 4)$. We remove $(1, 4)$ because it has lower persistence. This leaves us with the root branch $(0, 8)$.

The next thing we will present is the ascending filtration of the simplicial mesh and its extended persistence. The ascending filtration of the mesh is obtained by adding all vertices (and edges between them) one at a time in order of their height. We can obtain the persistence pairs by inspection because connected components correspond to persistence homology classes. Observe that a connected component is born at time 0 and at time 1. Those connected components merge at time 4. By enforcing the Elder Rule the component born at time 1 will die at time 4 and 0 will persist. The first persistence pair is $(1, 4)$. The following complexes of the filtration consist of a single connected component. Therefore the component born at time 0 is an essential class that never dies. We have to use extended persistence to obtain a pairing.

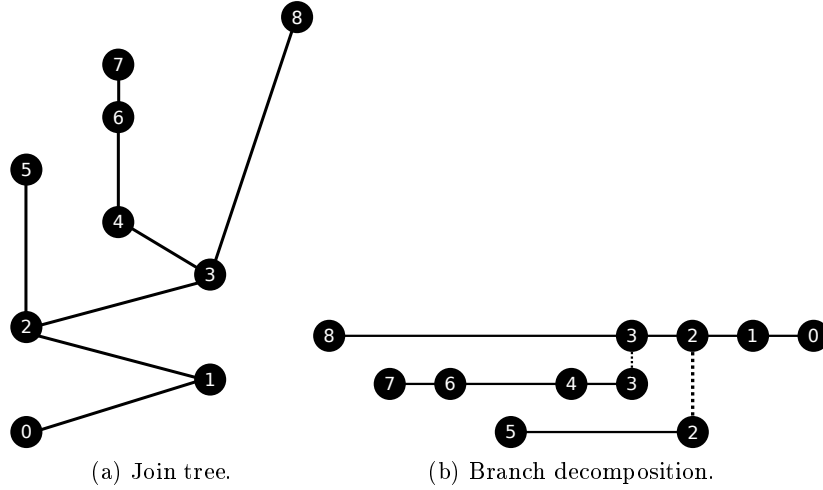


Figure 6.1: Branch decomposition of the join tree.

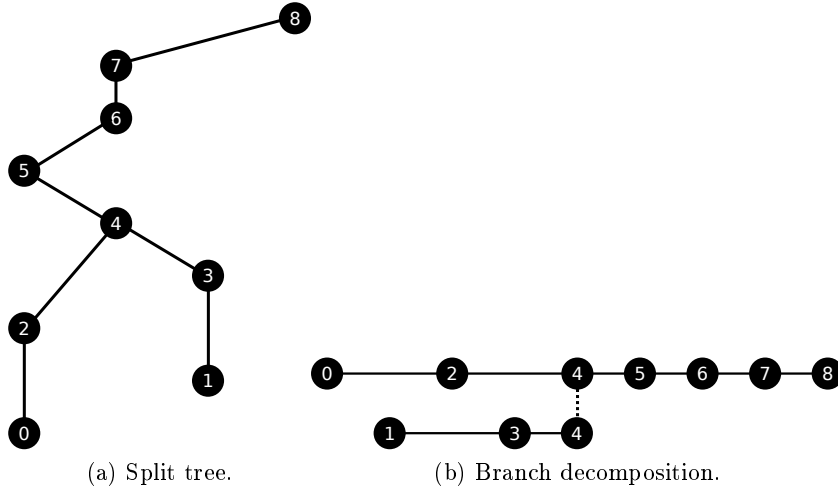


Figure 6.2: Branch decomposition of the split tree.

In order to do so we must find where in the relative filtration the homology class 0 dies. Let us first consider the last homology group of the ascending filtration $H_0(X_8) = H_0(X)$. Since X has one connected component $H_0(X_8) = \mathbb{Z}_2$. Now let us consider the next homology group in the extended filtration and obtain the induced map between them. The next homology group is $H_0(X, X^8) = H_0(X, \{8\})$. By the Excision Theorem we have that $H_0(X, \{8\}) = \tilde{H}_0(X/\{8\})$. We also know that a quotient space of a path connected topological space is path connected. Therefore $X/\{8\}$ has a single connected component and consequently $H_0(X/\{8\}) = \mathbb{Z}_2$. By the definition of reduced homology we know that $\dim(\tilde{H}_0(X/\{8\})) = \dim(H_0(X/\{8\})) - 1 = 0$. A zero dimensional vector space is the trivial vector space consisting only of the zero vector. The map $i_* : H_0(X_8) \rightarrow H_0(X, X^8)$ is the zero map because $H_0(X, X^8) = H_0(X, \{8\}) = \tilde{H}_0(X/\{8\}) = 0$. Therefore the homology class that persists in $H_0(X_8)$ dies in $H_0(X, X^8)$ and the final pair we obtain from extended persistence is $(0, 8)$. The two extended persistence pairs of the ascending filtration are $(1, 4)$ and $(0, 8)$.

We will compute the extended persistence pairs of the descending filtration in the same manner. Observe that one Figure 6.5 three 0th homology classes are born. They are born at times 0, 1 and 3. The component 1 dies at time 5 when it merges with the component born at

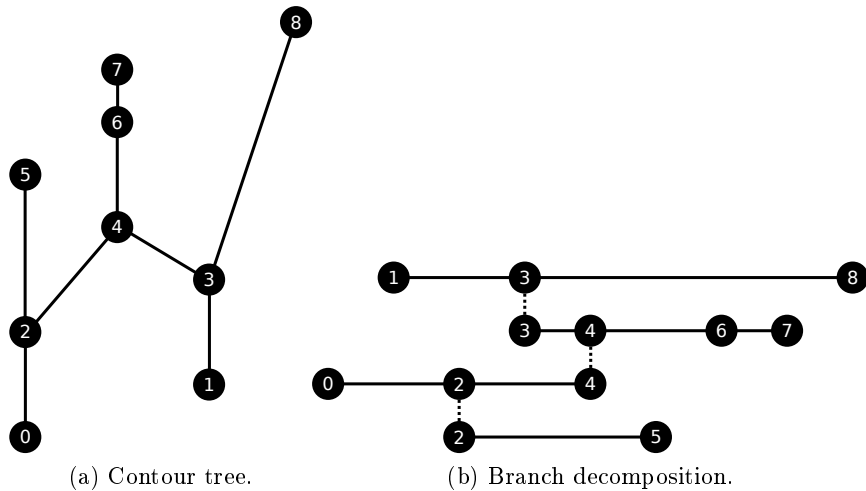


Figure 6.3: Branch decomposition of the contour tree.

time 0. The component 3 dies at time 6 when it is merged with component 0. The component 0 is the only essential homology class. We can compute its extended persistence pair in a completely analogous way to that of the ascending filtration. We obtain that the extended persistence pairs of the descending filtration are $(1, 5)$, $(3, 6)$ and $(0, 8)$. Those pairs correspond to vertex pairs $(3, 7)$, $(2, 5)$ and $(0, 8)$ because the vertex 3 is added at time 5, 7 is added at time 1, 2 at time 3, 5 at time 6, 0 at time 8 and 8 and time 0.

The barcode diagrams of the ascending and descending filtrations are presented on Figure 6.6.

We are now ready to answer the primary question set forth in the beginning of the section. Are the pairs of critical points produced by branch decomposition the same as the pairs produced by the extended persistent of the zeroth homology? They are not. In both the ascending and descending filtrations the global minimum 0 pairs with the global maximum 8. In the contour tree however there is no monotone path between them. Therefore they cannot be a pair in the branch decomposition.

To go a step further we will also examine the similarity of the join tree branch decomposition to the extended persistence of the descending filtration and the split tree branch decomposition to the extended persistence of the ascending filtration. We have depicted each of these side by side on Figure 6.7. The only change we have made is to relabel the time component of the descending filtration to correspond to the vertex that appears at that time.

We can see that in this example the two computations are equivalent. This does indeed make intuitive sense. The join tree captures topological information of how the connectivity of superlevel sets changes and so does the 0th extended persistent homology of the descending filtration. The same holds for the split tree and the 0th extended persistence of the ascending filtration. This is further supported by the fact that the book [22, p. 158] describes an algorithm for computing the 0th persistent homology that is almost identical to the algorithm for computing join/split trees. Whether they are equivalent latter remains to be proven rigorously in the general case.

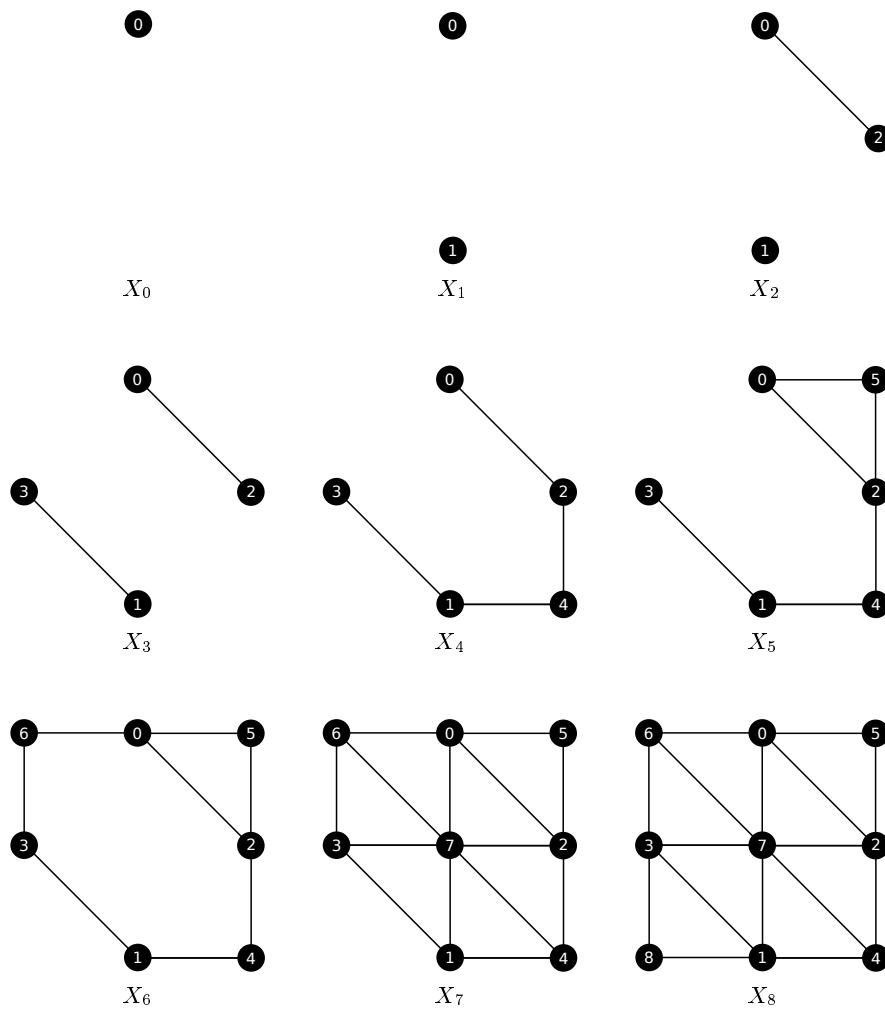


Figure 6.4: Ascending filtration of the dataset.

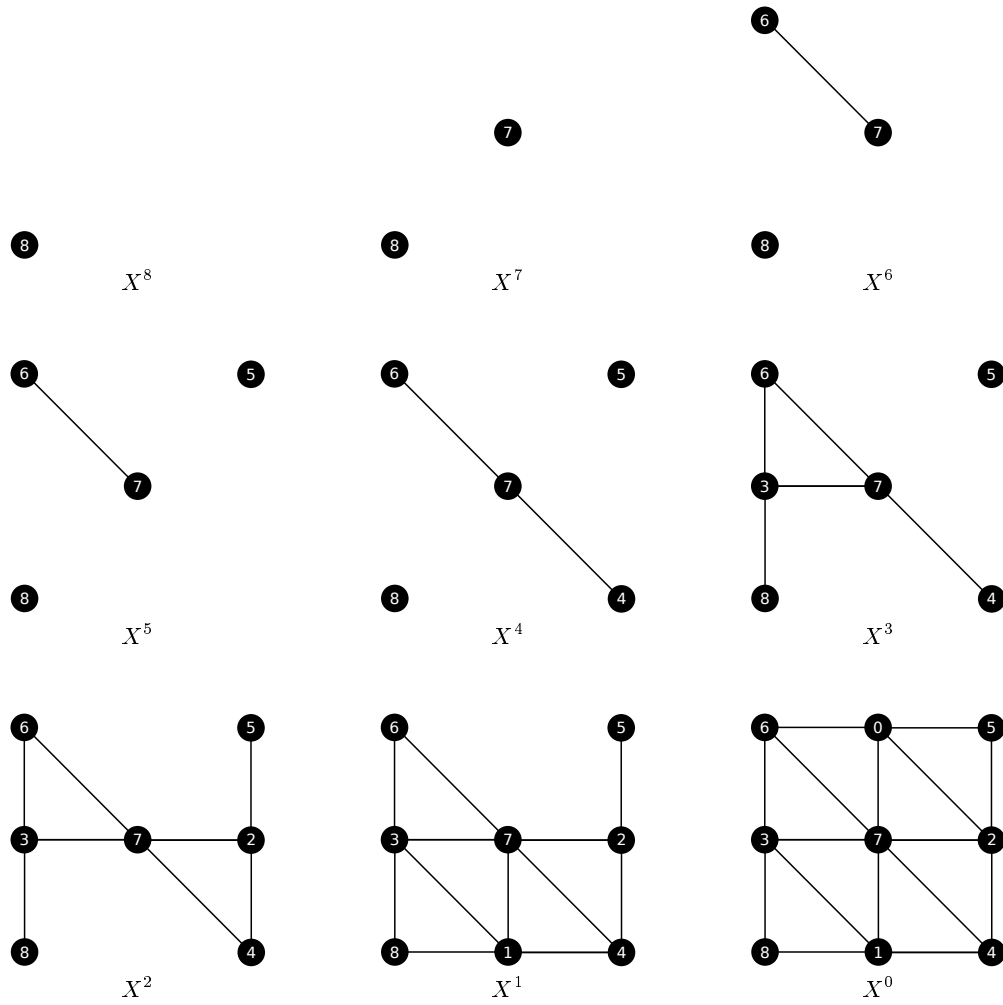


Figure 6.5: Descending filtration of the data set.

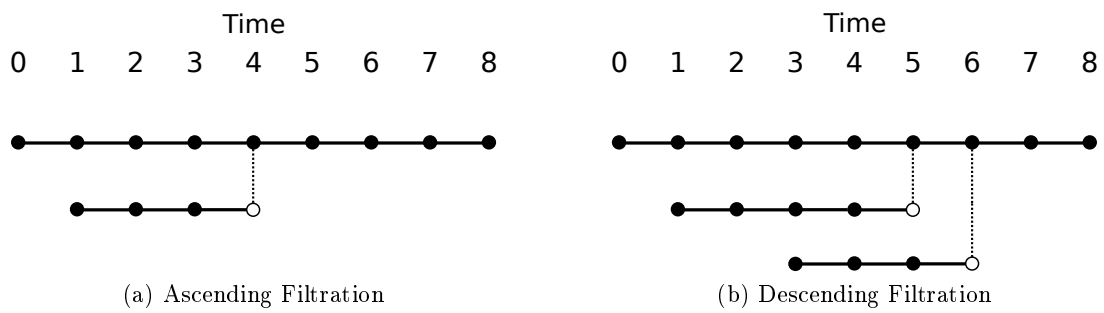


Figure 6.6: Barcode diagrams of the persistent pomology of the ascending and descending filtration.

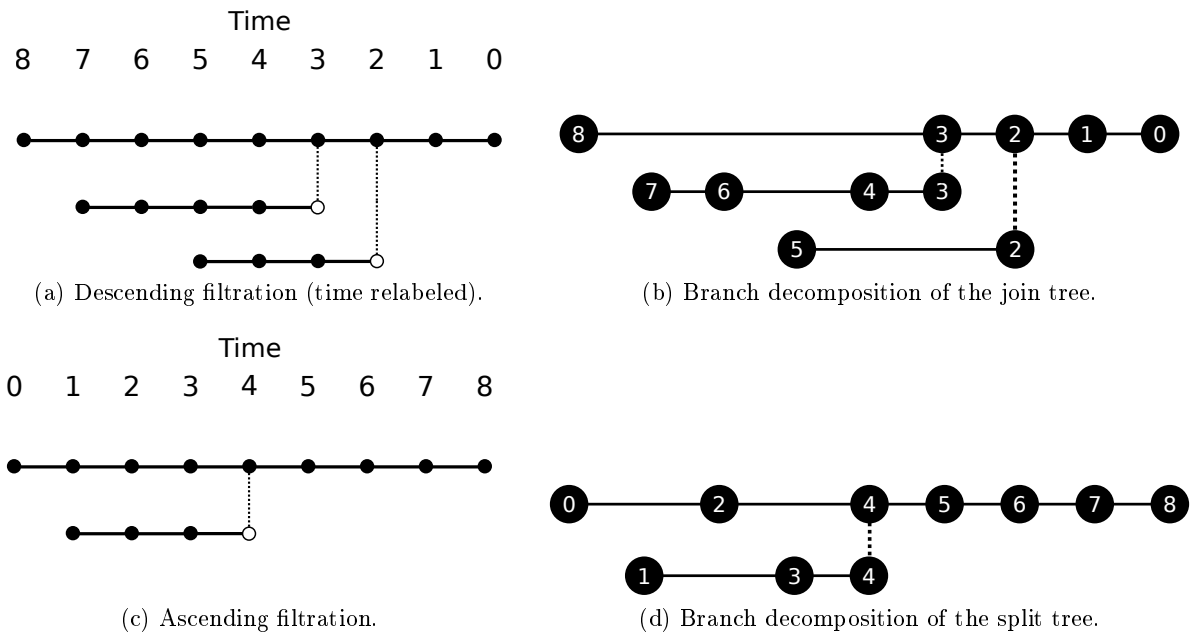


Figure 6.7: Branch decomposition of the join/split trees and extended persistence of the filtrations.

Chapter 7

Empirical Study

In this chapter we will supplement our theoretical investigation of the w-structures with an empirical study. The goal of this study is twofold. Firstly it is to verify the theoretical claims we have made on the correctness and running time of the w-diameter algorithms we developed. This will be done by implementing and testing them on a range of datasets. The second and most important goal of the empirical study is to analyse the w-structures that are present in contour trees of real life data sets using the w-diameter algorithms we have implemented.

7.1 Algorithm Implementations

For the purpose of conducting the empirical study we implemented all three w-diameter algorithms we developed in Chapter 4. Those are the brute force algorithm (NxBFS), the 2xBFS and DP algorithms. We implemented the brute force algorithm by running the modified version of BFS from 2xBFS from every vertex in the tree. The implementation of the 2xBFS algorithm was based entirely on the pseudocode we provided in Chapter 4. Initially we made a recursive implementation of the DP algorithm based on the pseudocode we provided. This approach was not efficient enough because the recursion added a substantial amount of overhead and slowed down computation on large data sets. We resorted to converting our recursive solution to an iterative one.

When converting a dynamic programming algorithm from a recursive to an iterative solution one first identifies all base case subproblems and solves them. Afterwards one works their way up to subproblems that depend only on the base case solutions and solves those. This process is repeated until all subproblems are solved. In our particular example the base cases are the leaves of the tree. The subproblems that depend on those are the vertices whose distance from a leaf is one. Once those are solved we move on to vertices whose distance from a leaf is two and so on.

To begin with we first have to root the tree using a standard Breath First Search. The root of the tree can be any vertex. Using the BFS we not only assigning parents to all vertices, but we also rank them based on their distance from the root. By processing vertices from furthest to the root we ensure that the subproblems of any vertex are solved for because its children have a bigger distance. In order to solve a subproblem at a vertex we can use the code we have provided in the backtracking part of the DFS in the pseudocode in Chapter 4.

Finally we also implemented the serial contour tree algorithm as it is described in [14]. The reason for doing so is that we needed to use it to compute the contour tree of test data and then execute the w-diameter algorithms on those contour trees. This required us to make some modifications to the output of the contour tree algorithm to better suit the expected input of the

w-diameter algorithms. All four algorithms we developed were written in C++ and their source code can be found in their GitHub repositories listed in Appendix D.

7.2 Data sets Overview

Before describing our testing methodology we will first elaborate on the types of data sets we will be using throughout our tests. The first type of data sets we will use are randomly generated height trees. In testing the correctness and running time of the w-diameter algorithms we would ideally like to run them on as many different data sets as possible in order to confirm our theoretical claims for all of them. Generating height trees randomly will allow us to produce new data sets for testing on demand. We will now describe the algorithm we implemented for generating random trees.

Starting with a disconnected graph G with n vertices we generate pairs of vertices u, v with random labels generated uniformly in the range $\{1, 2, \dots, n\}$. If adding the edge uv to the graph does not create a cycle we keep the edge. If it does create a cycle we discard it. We continue generating edges until G is fully connected. Upon reaching this point G will be a connected graph with no cycles. This is exactly the definition of a tree. In order to produce valid height trees we assign a random height to each of the vertices of the tree. In order to detect whether adding an edge produces a cycle we use the union-find data structure to keep track of which connected components vertices belong to. We add an edge between two vertices only when they belong to a different connected component and then merge the two components together.

The second type of data sets we will use is real life data taken from the GTOPO30 [5]. GTOPO30 is a digital elevation model of the world. The dataset is a two dimensional data grid containing the elevation of points on Earth with a resolution of approximately one kilometer. Due to the overwhelming size of the whole data set we have taken several smaller subsets of GTOPO30 provided to us by Dr. Hamish Carr.

The data sets are chosen based on their topographic complexity. All of them are both mountainous and lowland regions in Canada. The data sets we will use are named vanc (18x21), vancouverSWSW (25x49), vancouverSWNE (25x50), vancouverSWNW (25x50), vancouverSWSE (25x51), vancouverNE (49x99), vancouverNW (49x100), vancouverSE (50x99), vancouverSW (50x100), icefields (240x240), pukaskwa (551x1600) and gtopo30w020n40 (6000x4800). The vanc and all other vancouver data sets are taken from the North Shore Mountains that overlook Vancouver in British Columbia, Canada. The data set pukaskwa is taken from the Pukaskwa National Park located south of the town of Marathon, Ontario, Canada. The data set icefields straddles the continental divide. It is taken from Icefields Parkway in the heart of the Canadian Rocky Mountain. Finally gtopo30w020n40 is the data set that contains all other data sets.

7.3 W-detector Algorithms

In this section we will use our implementations of the Nx BFS, 2x BFS and DP algorithms to empirically test whether our claims on their correctness and running time are correct. The first

test we will present is on the correctness of all three algorithm. The main issue we encountered in this test is that all three algorithm solve a problem that to our knowledge has not been considered extensively. The only way to establish ground truth on their output is to manually inspect the w-diameter of a height tree and compare it with their output. This is neither reliable nor scalable to height trees with more than a few a few dozen vertices.

To overcome this issue we opted for using the output of the NxBFS algorithm as ground truth. The reason is that it is the most straightforward to implement and that its correctness is a trivial consequence of its formulation. We believe that this makes it the most reliable of the three. A further complication arises with the fact the NxBFS algorithm's running time is quadratic and not linear like 2xBFS and possibly DP. This leaves us unable to use this methodology for large enough trees where the NxBFS algorithm's computation simply scales to unreasonable time. This is why we have limited ourselves to only testing correctness for trees of up to 10,000 vertices. For this test we have used the following testing methodology:

- Generate a random tree.
- Run all three algorithms on the tree.
- Check whether the output of DP and NxBFS are the same and if the output of 2xBFS is within two of their output.

This methodology was used on one hundred trees of sizes 50, 100, 150, ..., 5500 resulting in 1000 individual tests all together. The output of all tests was in line with what we predicted in Chapter 3. The algorithms DP and NxBFS had identical output. A somewhat suprising result was that the 2xBFS algorithm did not output the wrong w-diameter on any of the tests.

The second test that was performed was on the running times of the algorithms. The algorithms were separated in two groups. The first group contained NxBFS and the second group contained 2xBFS and DP. They were separated because we expect the running time of NxBFS to be quadratic, 2xBFS to be linear, DP to be close to linear. We want the results within each group to be comparable. We tested the NxBFS algorithm on five randomly generated trees with sizes in the range $\{1000, 1500, \dots, 10000\}$. We avaraged the running time for each tree size. The results from testing the NxBFS algorithm were in line with what we expected. The running time the algorithm produces on Figure 7.1 fits a quadratic curve.

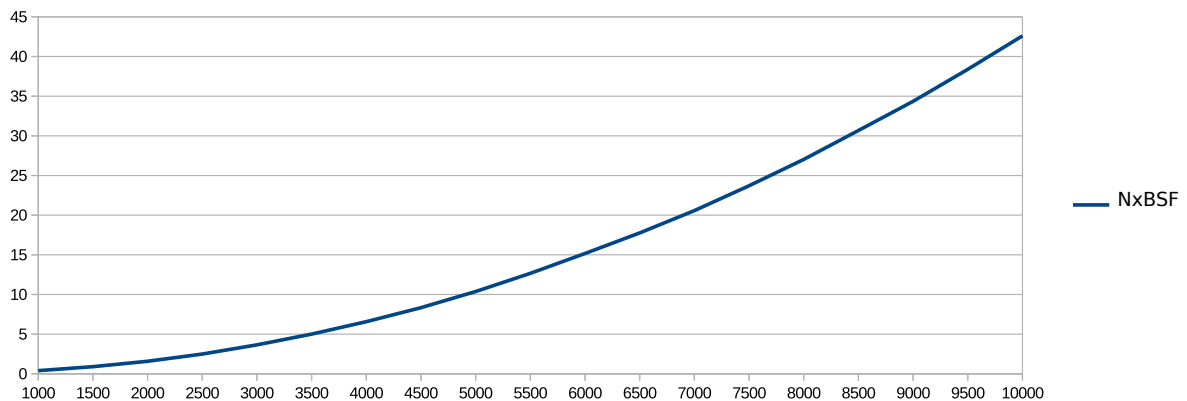


Figure 7.1: Running time of NxBFS on randomly generated trees.

The test of the running time of the 2xBFS and DP algorithms provided more insight. The test consisted of generating five trees per tree size in the range of $\{5000, 10000, 15000, \dots, 200000\}$. We ran both algorithms on all five trees for all tree sizes totalling 195 tests altogether. We plotted the average running time across all five trees per tree size on Figure 7.2. This chart shows that the performance of both 2xBFS and DP scales linearly with randomly generated height trees in the $[5000, 200000]$ vertex range.

This comes as no surprise as regards to the 2xBFS algorithm. We showed that its running time is linear. What is interesting to see is that the DP algorithm's performance scales linearly as well. We must note however that these tests use randomly generated trees and are a very limited sample so this trend may not hold for all possible inputs and especially for larger trees. Despite these reservations this test gives credibility to the claim we made in the Chapter 4 that the DP algorithm has potential for good practical performance despite its quadratic worst case time complexity.

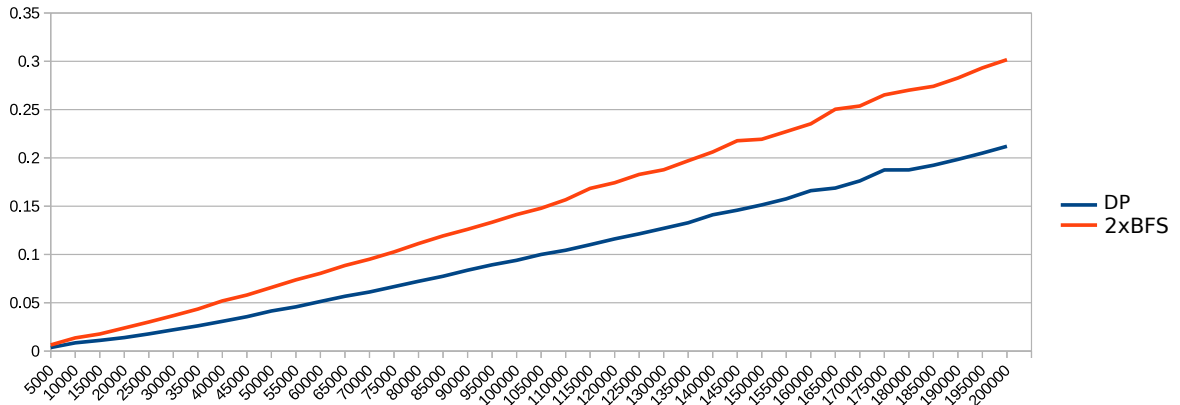


Figure 7.2: Running time of 2xBFS (blue) and DP (red) on randomly generated trees.

The next test we made was on the running time of the algorithms on the GTOPO30 data sets. We ran the tests for both the augmented and unaugmented contour trees and recorded how many times faster 2xBFS is than DP in the "Factor" column of the tables. The first test was with the augmented contour tree.

Dataset	Vertices	2xBFS	DP	Factor
vanc	378	0.000413	0.000497	1.20
vancouverSWSW	1225	0.000918	0.002317	2.52
vancouverSWNE	1250	0.000937	0.001858	1.98
vancouverSWNW	1250	0.000798	0.002454	3.08
vancouverSWSE	1275	0.001009	0.00204	2.02
vancouverNE	4851	0.003637	0.005493	1.51
vancouverNW	4900	0.002893	0.005612	1.94
vancouverSE	4950	0.002958	0.005863	1.98
vancouverSW	5000	0.002795	0.005658	2.02
icefield	57600	0.036430	0.066978	1.84
pukaskwa	881600	0.520608	0.962897	1.85
gtopo30w020n40	28800000	19.103233	34.427916	1.80

The results from this test are completely in line with what we obtained in Figure 7.2. The linear relationship between the running times of two still holds. DP is within the range of 1.2 - 3 times slower than the 2xBFS algorithm. In our next test we tested the same data sets, but we computed their unaugmented contour tree.

Dataset	Vertices	2xBFS	DP	Factor
vanc	29	0.000136	0.000057	0.42
vancouverSWSW	58	0.000372	0.000127	0.34
vancouverSWNE	148	0.000355	0.000237	0.67
vancouverSWNW	88	0.000412	0.000163	0.40
vancouverSWSE	109	0.000351	0.000193	0.55
vancouverNE	946	0.001888	0.001730	0.92
vancouverNW	911	0.001505	0.001292	0.86
vancouverSE	782	0.001466	0.001107	0.76
vancouverSW	380	0.001756	0.000802	0.46
icefield	7655	0.013704	0.010280	0.75
pukaskwa	65826	0.183262	0.100899	0.55
gtopo30w020n40	2436622	6.609592	3.742080	0.57

The results here are quite surprising and they required us to double check the test multiple times. On the outlook it seems that the two algorithms had exchanged their positions. The DP algorithm is 1.2 - 3 times faster than the 2xBFS algorithm on all data sets. The difference between the augmented and the unaugmented contour tree is that in the unaugmented contour tree all vertices of degree two are removed. One explanation for this is that such vertices may be processed faster by the 2xBFS algorithm and that the DP algorithm has some overhead associated with them. Further tests will be needed to determine what the cause of this is.

7.4 Dataset w-diameter analysis

In this section we will examine the w-structures present in real life data sets. Our goal is to demonstrate that they not only pose a theoretical difficulty, but also hinder practical performance. In this test we will execute our w-diameter algorithms on the mountain range data sets taken from GTOPO30. We will compare the w-diameters we obtain from the contour trees of the data sets with the diameters of the unaugmented and the augmented contour tree. We will demonstrate that the w-diameter of a contour tree is a better theoretical upper bound on the time complexity of the parallel contour tree algorithm than either of the two diameters.

Secondly we will compare the w-diameter of a contour tree with the number of iterations that are need to collapse the join and split trees. We hope to find a correlation between the two that would support our theoretical claim that it is the largest w-structure in a contour tree that prevents logarithmic collapse in the merge phase. We have taken the number of iterations needed to merge the join and split trees by compiling and running an implementation of the data-parallel contour tree algorithm based on [17] provided to us by Dr. Hamish Carr.

Dataset	2xBFS	DP	NxBFS	Aug Diameter	Diameter	Iterations
vanc	2	2	2	311	11	2
vancouverSWSW	2	2	2	845	17	3
vancouverSWNE	5	5	5	423	34	4
vancouverSWNW	3	3	3	712	23	3
vancouverSWSE	3	3	3	759	30	3
vancouverNE	4	5	5	1338	128	5
vancouverNW	5	5	5	1456	98	5
vancouverSE	6	6	6	1306	118	5
vancouverSW	4	4	4	1977	48	4
icefield	7	7	7	12280	886	6
pukaskwa	180	180	N/A	374866	1046	94
gtopo30w020n40	8	8	N/A	15766966	305290	8

We could not run the NxBFS algorithm on the largest data sets - pukaskwa and gtopo30w020n40 due to its slow poor performance.

The first inference we can make is about the pukaskwa data set. Both the w-diameter and number of iterations are drastically bigger than all of the other data sets. If logarithmic collapse was taking place in the merge phase of the construction of the contour tree of pukaskwa then we would expect that to take $\lceil \log_2(881600) \rceil = 20$ iterations. Instead it takes 94 iterations. This is consistent with the w-diameter of the data set. Indeed the w-diameter at 180 kinks is almost twice as big as the number of iterations 94. Consider that the algorithm can process exactly two branches on opposite sides of the w-diameter in a single iterations. It follows that at least $180/2 = 90$ iterations are needed for the full w-diameter to be collapsed.

Secondly, we can confirm that the w-diameter in almost all of the data sets (except for vancouverSWSW) is bigger than or equal to the number of iterations. This leads us to believe that the two may be correlated. In the case of pukaskwa we have already given an interpretation of this correlation. By that reasoning we would expect that in other data sets the w-diameter to be twice as much as the number of iterations. This is not the case and this may very well be due to how different w-structures interact with one another in the merge phase. This is something we have not investigated as we only record the size of the largest w-structure.

Finally we turn our attention to the diameter of the augmented and unaugmented contour trees. The parallel contour tree algorithm currently uses those as an upper bound on the time complexity of the merge phase [17]. We have already shown theoretically that the w-diameter of a height tree is necessarily smaller than its diameter. This test demonstrates how big the difference between the two can be in practice. In the most extreme example, that of gtopo30w020n40, the w-diameter of the augmented contour tree is 1,970,870 times smaller than its diameter. If the w-diameter of that data sets were equal to the actual diameter, then we would expect the merge phase to take a far larger number of iterations and severely limit the available parallelism in it.

Chapter 8

Conclusion

In this dissertation we examined a particular tree structure that hinders the parallel algorithmic performance of the state of the art data-parallel contour tree algorithm. These structures are long paths in height trees with a characteristic zigzag pattern. We call them w-structures. In order to better understand them we developed three algorithms for the detection of the largest w-structure in a contour tree. We proved those algorithm are correct and showed formal bounds on their time and space complexity. We implemented these algorithms and used them to establish the existence of w-structures in contour trees of real life data. An empirical study releaved that that the largest w-structure in a height tree does impact available parallelism in one of the two phases of the data-parallel contour tree algorithm.

In addition to this we explored the use of a tool from Topological Data Analysis called Persistent Homology. Persistent homology is a general framework for topological simplification. Our interest was in whether it is equivalent to a contour tree specific tool for topological simplification called branch decomposition. We demonstrated that they are not equivalent by computing both on a counter example based on the w-structures.

8.1 Personal Reflection

The most difficult part of the project was learning the prerequisite mathematics. Those spanned the fields of Topology, Algebraic Topology and Computational Algebraic Topology. I had covered some of these topics in my second semester with a module on Topology that included an introduction to Algebraic Topology. Throughout the summer the most time consuming part was to learn Homology and to then learn how to apply it via persistent homology and by extension extended persistence. The main difficulty was in making sense of the all the moving components that enable extended persistence. Those are Morse Theory, Point Set Topology, Homology, Cohomology, Spectral Sequences and the Poincare-Lefschetz duality. In the end I did not have to use all of them for the specific problem I was trying to solve. Despite this I still had to learn how the ideas borrowed from those fields work and relate to one another and I had to pick out exactly the ones I needed for my specific proofs.

My approach to learning those fields was not optimal. I overcommitted a lot of my time and energy by trying to immediately learn how extended persistence is computed. Without sufficient prerequisite knowledge this attempt was futile. Henceforth I decided to make my approach more systematic and disciplined. I did this by outlining all of the different ideas, definitions and theorems that build the foundation of extended persistence. I used this to go through all of them in a bottom up fashion. I went through the relevant books and papers slowly and carefully and I made sure I can convince myself of the theory by creating small scale examples and giving myself

simple problems to prove. Overall I am satisfied with the results I have obtained. This gave me valuable experience in teaching myself new mathematics and approaching novel research areas.

What I am most proud with in this dissertation is creating and implementing the w-diameter algorithms. I started off with a well-defined problem that had not been solved before and I had to come up with an algorithm for it. The first thing I did was to try and come up with definitions that capture the problem statement and allow me to work with it in a formal setting. It took me some experimentation to come up with the characterisation of w-paths via kinks and to realise this characterisation can be used as a metric much like path length. This helped me recognise that it may be possible to modify existing tree diameter algorithms.

The first algorithm I worked on was the 2xBFS algorithm. I started off with just the idea and first implemented it to test whether it holds in practise. Upon obtaining satisfactory results I began to look for ways to modify the proofs of correctness of the original tree diameter algorithm. My initial plan was to dissect proofs of tree diameter algorithms and see exactly which of their components I have to modify. The most challenging aspect of proving the correctness of 2xBFS was in discovering all of its pathological cases and working on adjusting the proof to accomodate them.

Initially I did not have a clear idea how I can implement the DP algorithm. Through trial and error on numerous small scale example height trees I came up with the theoretical formulation of the solution. For this algorithm my mathematical proof preceded my implementation. My initial proof however was not correct. The way I realised this was by implementing it and observing a discrepancy in the output. I used that discrepancy to trace my mistake and correct my proof.

The part of the dissertation I believe I could have developed more is the empirical study. The empirical study was supposed to be a more central topic, in the case that I failed to produce meaningful results with extended persistence. After committing more time than I had planned on the w-diameter algorithms and extended persistence I had already covered enough material and I had little time to produce a more detailed and insightful empirical study.

Overall I believe that the project was successful. I accomplished all of the tasks I set out to do. In the end it turned out to be more theoretical than I had originally planned. I am glad that this is the case. This gave me the opportunity to advance my mathematical and analytical skills. It has also given me the foundation to continue my research in new research directions.

8.2 Future Work

In this section we will present a number of possible direction for future work. They will be split in two parts. The first part will be related to the empirical study of the w-structures and second part to the further exploration of the use of extended persistence in contour tree simplification.

One obvious starting point would be to extend the empirical study by obtaining more real life data to analyse. Before doing so however we would need to develop better tools to analyse the w-structures in that data. One way to advance our w-diameter algorithms would be to integrate them with the contour tree algorithm and use them to analyse how different w-structures interact

with one another in the merge phase. This would require us to modify our algorithms to find more than just the w-diameter of the tree. It would be most useful to use them to obtain the first few biggest w-structures.

Another useful direction we can take is to try and develop an algorithm that can compute the w-diameter from input data directly, without having to compute the contour tree first. One hope we have is that this will enable us to spot patterns in data that correspond to w-structures in the contour tree and use this prior knowledge in the contour tree construction to obtain better parallel performance. By spotting these patterns in data we would hopefully be able to categorize them. This would ideally lead to a formal proof that limits the number of patterns in data that produce large w-structures.

Finally we would like to propose an idea for a future direction a purely theoretical line of research can take. We saw that we can express the branch decomposition of the join and split tree with the extended persistence of the ascending and descending filtration. But is it possible to also express the branch decomposition of the contour tree with some other filtration?

Consider for example a simplicial mesh M and its level sets M_i . Homology gives us tools to identify the connected components of the M_i by computing their 0th homology. In order to track how the homology classes evolve as we vary the parameter i we would need a way of relating the homology classes of different level sets. This can take the form of a sequence:

$$\dots \rightarrow H_0(M_i) \rightarrow H_0(M_{i+1}) \rightarrow \dots \rightarrow H_0(M_j) \rightarrow H_0(M_{j+1}) \rightarrow \dots$$

The question is what the linear maps between the homology groups would be. We cannot induce them via inclusion maps because the level sets are not subsets of one another. A research direction in this area would be to find another way of obtaining simplicial maps between the level sets. Obtaining such maps will allow us to induce linear maps between their respective homology groups.

References

- [1] Double breadth first search tree diameter algorithm. <http://courses.csail.mit.edu/6.046/fall101/handouts/ps9sol.pdf>. Accessed: 2017-09-11.
- [2] Double breadth first search tree diameter algorithm implementation. <http://www.geeksforgeeks.org/longest-path-undirected-tree/>. Accessed: 2017-09-11.
- [3] Dynamic programming tree diameter algorithm. <https://users.cs.duke.edu/~ola/courses/cps100spr96/tree/trees.html>. Accessed: 2017-09-11.
- [4] Dynamic programming tree diameter algorithm implementation. <http://www.geeksforgeeks.org/diameter-n-ary-tree/>. Accessed: 2017-09-11.
- [5] Gtopo30 data set. <https://lta.cr.usgs.gov/GT0P030>. Accessed: 2017-09-11.
- [6] P. K. Agarwal, H. Edelsbrunner, J. Harer, and Y. Wang. Extreme elevation on a 2-manifold. In *SCG '04 Proceedings of the twentieth annual symposium on Computational geometry*, pages 357–365. ACM, 2004.
- [7] S. Axler. *Linear Algebra Done Right*. Springer, 3 edition, 2015.
- [8] T. Banchoff et al. Critical points and curvature for embedded polyhedra. *Journal of Differential Geometry*, 1(3-4):245–256, 1967.
- [9] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno. Reeb graphs for shape analysis and applications. *Theoretical Computer Science*, 392(1-3):5–22, 2008.
- [10] P.-T. Bremer, B. Hamann, H. Edelsbrunner, and V. Pascucci. A topological hierarchy for functions on triangulated surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):385–396, 2004.
- [11] G. Carlsson. Topology and data. *Bulletin of the American Mathematical Society*, 46(2):255–308, 2009.
- [12] H. Carr. Efficient generation of contour trees in three dimensions. Master’s thesis, University of British Columbia, 2000.
- [13] H. Carr, Z. Geng, J. Tierny, A. Chattopadhyay, and A. Knoll. Fiber surfaces: Generalizing isosurfaces to bivariate data. In *Computer Graphics Forum*, volume 34, pages 241–250. Wiley Online Library, 2015.
- [14] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24(2):75–94, 2003.
- [15] H. Carr, J. Snoeyink, and M. Van De Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, 43(1):42–58, 2010.
- [16] H. Carr, J. Tierny, and G. Weber. Pathological and test cases for reeb analysis.

- [17] H. A. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable smp contour tree computation. In *Large Data Analysis and Visualization (LDAV), 2016 IEEE 6th Symposium on*, pages 75–84. IEEE, 2016.
- [18] D. Cohen-Steiner, H. Edelsbrunner, and J. Harer. Extending persistence using poincaré and lefschetz duality. *Foundations of Computational Mathematics*, 9(1):79–103, 2009.
- [19] M. L. Connolly. Shape complementarity at the hemoglobin $\alpha 1\beta 1$ subunit interface. *Biopolymers*, 25(7):1229–1247, 1986.
- [20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press Cambridge, Mass, USA, 2001.
- [21] H. Edelsbrunner and J. Harer. Persistent homology-a survey. *Contemporary mathematics*, 453:257–282, 2008.
- [22] H. Edelsbrunner and J. Harer. *Computational Topology, An Introduction*. American Mathematical Society, 1 edition, 2013.
- [23] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 454–463. IEEE, 2000.
- [24] R. Ghrist. Barcodes: the persistent topology of data. *Bulletin of the American Mathematical Society*, 45(1):61–75, 2008.
- [25] R. Ghrist. *Elementary Applied Topology*. Createspace, 1 edition, 2014.
- [26] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 1 edition, 2002.
- [27] D. Kozlov. *Combinatorial Algebraic Topology*, volume 21 of *Algorithms and Computation in Mathematics*. Springer, 1 edition, 2008.
- [28] C. Li, M. Ovsjanikov, and F. Chazal. Persistence-based structural recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1995–2002, 2014.
- [29] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012.
- [30] Y. Matsumoto. *An Introductino to Morse Theory*, volume 208 of *Translation of Mathematical Monographs*. American Mathematical Society, 1 edition, 2002.
- [31] B. Mendelson. *Introduction to Topology*. Dover, 3 edition, 1990.
- [32] J. Milnor. *Morse Theory*. Princeton University Press, 1 edition, 1963.
- [33] D. Morozov and G. Weber. Distributed merge trees. In *ACM SIGPLAN Notices*, volume 48, pages 93–102. ACM, 2013.
- [34] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2004.

- [35] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. Multi-resolution computation and presentation of contour trees. In *Proc. IASTED Conference on Visualization, Imaging, and Image Processing*, pages 452–290, 2004.
- [36] Y. Shi, J. Li, and A. W. Toga. Persistent reeb graph matching for fast brain search. In *International Workshop on Machine Learning in Medical Imaging*, pages 306–313. Springer, 2014.
- [37] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3d in $o(n \log n)$ steps. In *Proceedings of the fourteenth annual symposium on Computational geometry*, pages 68–75. ACM, 1998.
- [38] M. Van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the thirteenth annual symposium on Computational geometry*, pages 212–220. ACM, 1997.
- [39] S. K. Verovšek and A. Mashaghi. Extended topological persistence and contact arrangements in folded linear molecules. *Frontiers in Applied Mathematics and Statistics*, 2:6, 2016.
- [40] S. Weinberger. What is... persistent homology? *Notices of the AMS*, 58(1):36–39, 2011.
- [41] A. J. Zomorodian. *Topology for Computing*. Cambridge University Press, 1 edition, 2009.

Appendices

Appendix A

Ascending Filtration of the Contour Tree

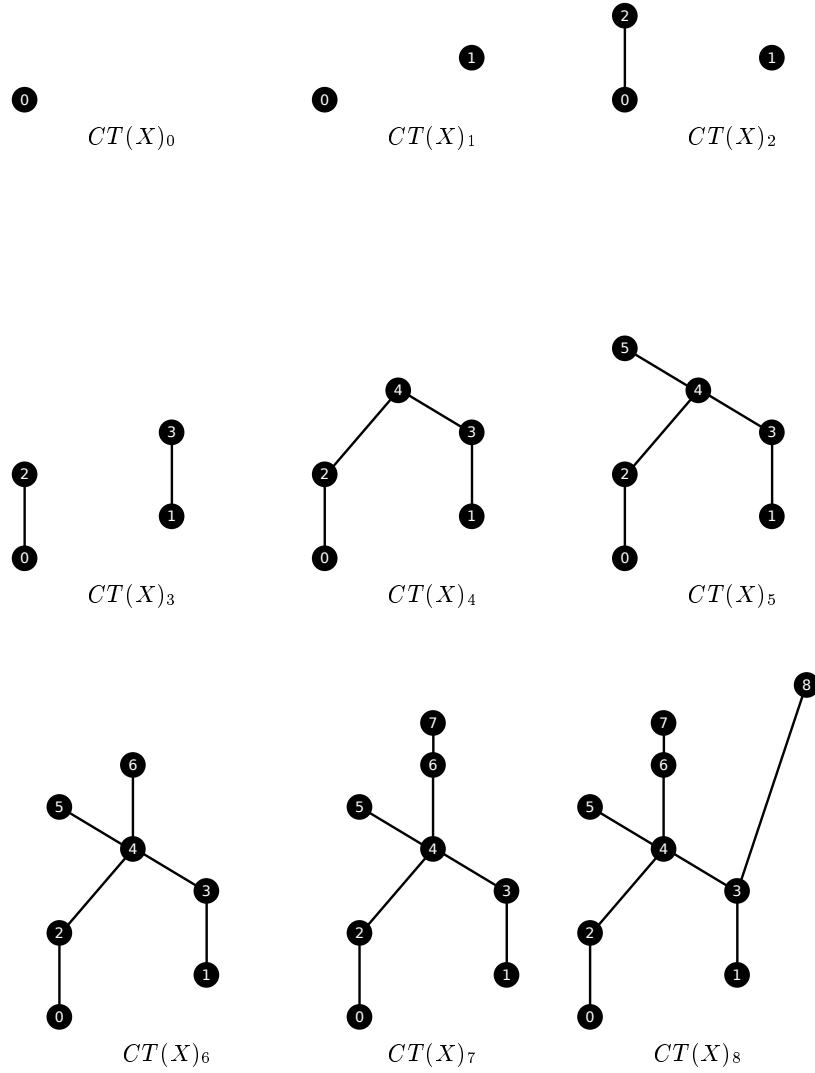


Figure A.1: Ascending filtration of the contour tree from Figure 3.2 b.

Appendix B

Descending Filtration of the Contour Tree

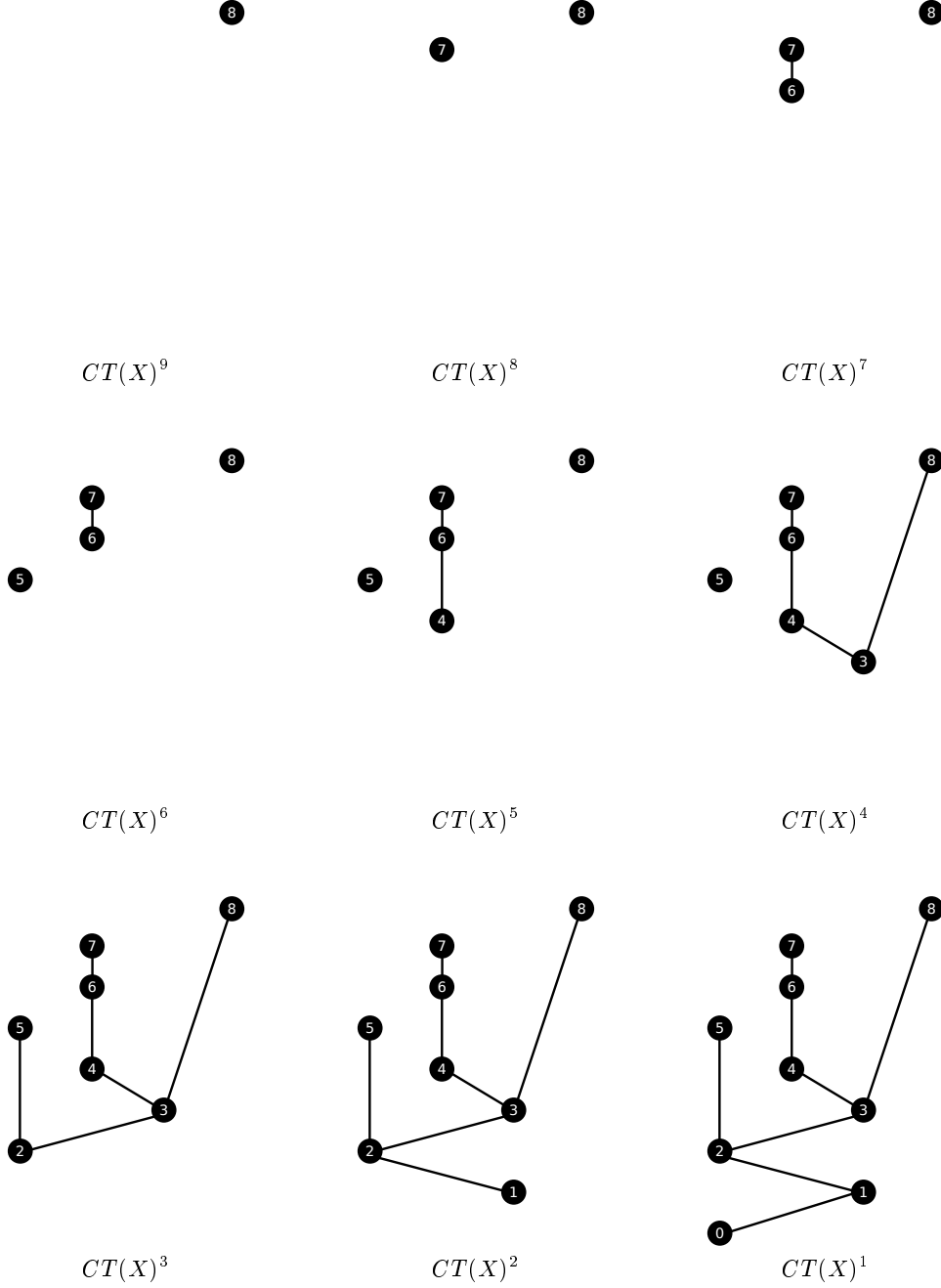


Figure B.1: Descending filtration of the contour tree from Figure 3.2 b.

Appendix C

Additional Proofs

Lemma 9. *In a tree with no vertices of degree two at least half of the vertices are leaves.*

Proof. Let $T = (V, E)$ be a tree with no vertices of degree two and let $L \subseteq V$ be the set of all leaves. As all leaves have degree one we have that $L = \{u \in V : d(u) = 1\}$. Furthermore for any tree we know that $|E| = |V| - 1$. Let us now use the handshake lemma:

$$\sum_{u \in V} d(u) = 2|E| = 2(|V| - 1) = 2|V| - 2.$$

We will not separate the sum on the leftmost hand side of the equation in two parts. One for the vertices in L and one for the vertices in $V \setminus L$.

$$\sum_{u \in L} d(u) + \sum_{u \in V \setminus L} d(u) = 2|V| - 2.$$

All the vertices in L are leaves. By definition the degree of a leaf is one. Therefore $\sum_{u \in L} d(u) = |L|$. This leads us to the following:

$$|L| + \sum_{u \in V \setminus L} d(u) = 2|V| - 2$$

$$|L| = 2|V| - 2 - \sum_{u \in V \setminus L} d(u).$$

There are no vertices in T of degree two and all vertices of degree one are in L . This means that all vertices in $V \setminus L$ have degree at least three. We can conclude that:

$$\sum_{u \in V \setminus L} d(u) \geq \delta(T - L) \cdot |V \setminus L| = 3(|V| - |L|)$$

Combining this with the previous equation we obtain that:

$$|L| \leq 2|V| - 2 - 3(|V| - |L|)$$

$$|L| \leq 2|V| - 2 - 3|V| + 3|L|$$

$$-2|L| \leq -|V| - 2$$

$$|L| \geq \frac{|V|}{2} + 1$$

Which is exactly what we set out to prove.

□

Appendix D

Github Repositories

DP Algorithm - <https://github.com/famouscake/w-detector>

2xBFS Algorithm - <https://github.com/famouscake/w-detector>

Contour Tree Algoritm - <https://github.com/famouscake/ContourTree>

Appendix E

Ethical Issues Addressed

E.1 Data Sources

For this project data sources were required to compute contour trees. The data sources that were used are publicly available on the Internet. They did not require any special permissionssions to use.

E.2 Software

All software that was used is publicly available on the internet and free. It has been referenced and acknowledgement in the text.