School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Something W this way comes

Petar Hristov

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

<Session>

The candidate confirms that the following have been submitted.
<As an example>

Items	Format	Recipient(s) and Date
Beliverable 1, 2, 3	Report	SSO (DD/MM/YY)
Participant consent forms	Signed forms in envelop	SSO (DD/MM/YY)
Deliverable 4	Software codes or URL	Supervisor, Assessor (DD/MM/YY)
Deliverable 5	User manuals	Client, Supervisor (DD/MM/YY)

Type of project: ____________________________

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 Tsolova for supporting both morally and emotinally and my good friends Atanas Angelov and Ishan Lee without which this dissertation would probably have been finished faster.

Contents

List of Figures

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 [32] and Topological Data Analysis [3]. 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 [30, 7], visualisation [26, 16, 15], medical imaging [31] and computer vision [27, 6].

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 [5]. 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 [13]. 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 [14]. Our goal it to understand how and why these w-strucures appear in contour trees of real life data. We will acomplish 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 [25]. 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 [8]. We will pose and answear the question of whether the two approaches are equivalent. A counterexample based 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 compating 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 artifically generated and on real life data sets. We will use those algorithms to demonstrate that these 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 posses 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 [21].

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 $x = (x_1, x_2, ..., x_n)$ 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 $f : X \to Y$ 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 then. 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 $f : [0, 1] \to X$ 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 on 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 suspace of X. Wwe 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 $\pi : X \to X/\sim$ 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 $\pi : X \to X/\sim$. An important example 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 they cannot be properly embedded in two or three dimensional Eucledian 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 [20, 22]. 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 ourself to a special class of differentiable functions called Morse functions.

Definition 8. *A function $f : M \to \mathbb{R}^n$ 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 topoogy 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 $f : M \to \mathbb{R}$ 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 ensures 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 dissapear 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 $f : X \to \mathbb{R}$ 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 though of as a discrete graph. To do se 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 begining 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 [?]. 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 Comlexes 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 [] where we will see how the concept of the Euler Characteristic can be generalised to the field of Algebraic Topology called Homology.

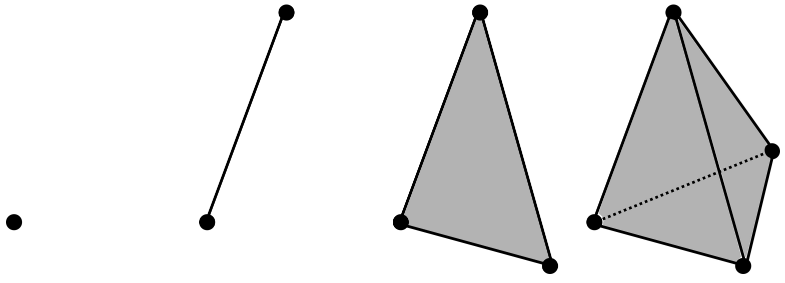
2.3.1 Simplicial Complexes

Simplical Complexes are the 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 [9].

Definition 11. *Let $\{v_0, ..., v_k\}$ 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 $\sum_{i=0}^{n} \lambda_i = 1$.*

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

Definition 12. *Let $\{v_0, ..., v_k\}$ 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 $[v_0, ..., v_n] \subset \mathbb{R}^{k+1}$*



(a) Vertex (b) Edge (c) Triangle (d) Tetrahedron

Figure 2.1: Simplices 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 subsets of its points. For example the faces of the tetrahedron are the four triangles, six edges and four vertices. To construct a simplical 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 simplical complex K is a finite collection of simplices such that if τ is a simplex K then all faces of τ must be simplicies in K. Furthermore the intersection of two simplicies in K is either empty or a common face of both.*

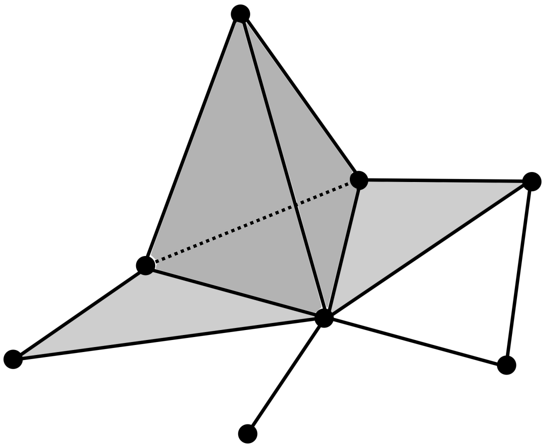


Figure 2.2: A simplicial comlex.

We obtain the topology of a simplicial complex by embedding it in Euclidean space and considering its subspace topology as a subet. After formalising the concept of a simplical 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 simplical complexes through a generalisation of counting [12]. 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 simplical complexes as as a infinite alternating sum 3-dimensional simplices, 4-dimensional simplices and so on.

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

where all k_i is the number of i-dimensional simplicies for $i \in \mathbb{Z}^+$ and $k_j = 0$ for i 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 disertation 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) [?]. 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 neighbouhood of u or all the vertices u is adjacent to. We will the notation $d(u, v)$ wherer $u, v \in V(G)$ for the length of the shortes 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 whoose 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.

In Chapter [] 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 [].

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). 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 [] 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 BSF are linear [?] and therefore the space and time complexity of this algorithm are linear as well. This follows from that fact that the algorithm consists of running BFS two consecutive times.

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

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 then 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, of the tree 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 sumamrised by he following formula.

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

The first terms describes the case when the longest path is contained entirely in one of the subtrees rooted at a child of s. The second terms combines the two longest height paths in two distinct children u, v of s and adds an aditional 2 to account for the edges us and vs. The last term is given in the case where s has exactly on 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) 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 [].

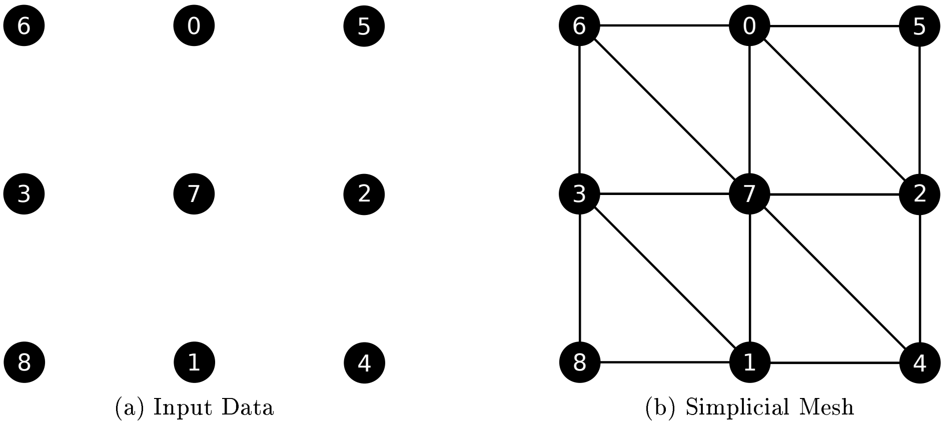
Chapter 3

Contour Trees

The Reeb Graph of a contractable topological space is connected and acyclic [9]. This allows us to define a special case of the Reeb Graph called 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 some graph theoretical properties of contour trees and then demonstrate how those are used in the construction of the algorithms. We will conclude with a discussion of a particular pathological case that causes poor performance in the parallel contour tree algorithm. In addition to this we will include the extra topic of contour tree simplification. Simplification is the process of identifying and removing parts of the contour tree that are not topologically significant.

3.1 Typical Data

Many scientific and medical applications require the sampling of points from a bounded area or volume in two or three dimensional Euclidean space [4]. 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 simplicies are added to completely fill the space between them (see Figure []). The resulting data structure we will call a simplical mesh [4].



(a) Input Data

(b) Simplicial Mesh

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

The values of the approximation function at the simplicies are obtained via linear interpolation between the verticies of each simplex. As long as the original values we have sampled are unique it can be shown that the resulting linear interpolation function is a Morse function and that the critical values are critical points are the vertices of the mesh [2].We will demonstrate in the following section how this crucial property enables efficient computation.

3.2 Algorithms for Computing Contour Trees

The first efficient algorithm for constructing contour trees [19] is due to Van Kreveld et al. Its running time is $O(NlogN)$ on two dimensional domains and $O(N^2)$ in higher dimensions where N is the number of triangles in the simplicial mesh. Tarasov and Vyalyi [29] extended this algorithm to work in time $O(NlogN)$ in 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. Shortly after Carr at. al [5] introduced an algorithm with running time $O(nlogn + N\alpha(N))$ where n is the number of vertices in the simplicial mesh and α is the notoriously slow growing inverse Ackerman function. This algorithm has the advantage that it 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 accomodate the distributed [23, 24] and shared memory parallelism paradigms [13, 28]. The focus of this dissertation we will be one of the latest devlopments in the creating a data parallel shared memory algorithm for contour tree computation [13]. Before introducing how that algorithm operates and one of the issues related to its parallel performance we will first give a more detailed overview on the most established serial algorithm [5] which the data parallel one is based on. In order to talk about any of two algorithm however we must establish some notation and define height graphs and trees as they are defined in [4].

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 our target application domain. 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 we can consider their 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, $V(G) = \{v_1, v_2, ..., v_n\}$ where $h(v_1) < h(v_2) < ... < h(v_n)$. 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 the graph is a sequence of vertices $(u_1, u_2, ..., u_k)$ where $u_i \in V(G)$ for $i \in \{1, 2, ..., k\}$

and $u_iu_{i+1} \in E(G)$ for $i \in \{1, 2, ..., k-1\}$. Furthermore a path in a height graph is ascending whenever $h(u_1) < h(u_2) < ... < h(u_k)$. Conversely 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 with higher value. It is denoted as $\delta^+(v) = \left|\{u \in N(v) : h(u) > h(v)\}\right|$.*

The down degree of v is defined analogously as $\delta^-(v) = \left|\{u \in N(v) : h(u) < h(v)\}\right|$. In the context of height trees the definitions of up and down degrees of a vertex allows 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 chapter how differentiating between the two types of leaves is used in constructing the contour tree.

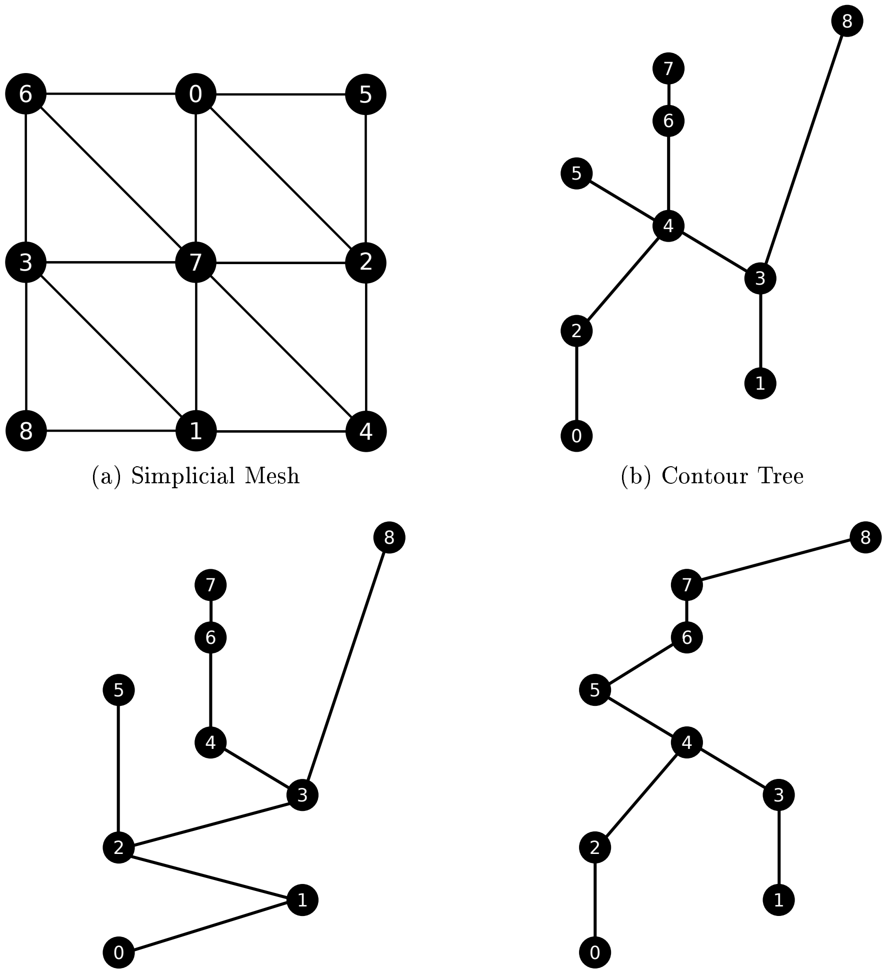
3.4 Join and Split Trees

The contour tree contains information for two types of events - joining and splitting of contours. Based on this we can take a contour tree and derive from it two other height trees that each contain the information of joining and splitting separately. We will call these 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. See Figure []. 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. The two are symmetric in that the join tree of the function f is isomorphic to the split tree of the negative of the function $-f$. You can find an example of the join and split trees of Figure [] on Figure [].

The reason we would like to study join and split tree is that the contour tree can be reconstructed from them. The core idea of the algorithm we will present is that we can derives the join and split trees directly from the simplicial mesh and then combine them to obtain the contour tree. Let us first examine how join and split trees are computed from the mesh. We will describe for the process solely for the join tree. The computational for the split tree is completely analogous. We first need the following definition.

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

Let M be our simplicial mesh and $h : M \to \mathbb{R}$ be the interpolation function defined on it. To construct the join tree we are going to have to keep track of which components merge together in the sublevels sets of h. We will consider all sublevel sets $M_t = h^{-1}((-\infty, t]) = \{x \in M : h(x) \in (-\infty, t]\}$ 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



(a) Simplicial Mesh

(b) Contour Tree

(c) Join Tree

(d) Split Tree

Figure 3.2: The Simplicial Mesh, Join and Split Trees and Contour Tree.

of the sublevel sets changes as the parameter t is increased. The connectivity of sublevel sets changes either at local minima where a new component is created or a saddle point that merges two 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 sublevel set $h^{-1}((-\infty, t])$ 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 [9]. Furthermore if $M_m = M$ is the last sublevel 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 [5] for further implementational details. The algorithm works by considering the vertices of the simplicial mesh in ascending order of height. If the current vertex is a local minimum we directly add it as a vertex 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. This is exactly there the $\alpha(n)$ term in the time complexity of the algorithm comes from.

Not all vertices of the mesh will be in the join tree. Only those which correspond to local minima 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 $\{v_1, v_2, ..., v_n\}$ 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 $i \in \{1, 2, ..., n\}$ 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 kth step we subdivide the edge $v_{k-1}b$ and label the new vertex as .

* Show some pretty pictures join, aug join, split, aug split, contour, aug contour*

The procedure of augmentation can be applied to the 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 second step of the algorithm is to combine the join and split trees to produce the contour tree. We will in fact be combining the augmented join tree with the augmented split tree to obtain the augmented contour tree. Removing the augmentation of the contour tree is then left as an optional final step. The first step in merging the two it 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 [4].

Definition 17. *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 it's 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.*

We will make an analogous definition in the case of down leaves and their adjacent edges.

Definition 18. *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 it's 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 it's adjacent edge in the split or join tree. Another property [4] tells us that if we perform vertex contraction on v (remove v and form clique from its neighbours) 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 reapeat this process until we have removed all vertices. We are allowed to do so because removing all leaves in a tree leaves the tree with at least one leaf if it is not empty where the algorithm will terminate. For a detailed example of this process we refer the reader to [5].

3.5 Serial Algorithm

The Serial algorithm for the construction of the contour tree is a summary of the results we outlined in the previous section. It works as follows:

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

Step 2. Compute the Join and Split Tree of the mesh.

Step 3. Iteratively remove leaves from the Join and Split tree and add them to the Contour Tree until the Join and Split trees are empty.

Step 4. Remove regular vertices of the contour tree if necessary.

3.6 Parallel Algorithm

The data parallel contour tree algorithm [13] is largely based on the theory we have established so far. 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 involved and completely separate from the problem we aim to address. We will however describe in detail how the merge phase is parallelised.

In general 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 and that is costly in terms of performance. If we observe the merge phase of the serial algorithm we will notice that removing a leaf 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 all 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. Consequently 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. If we can ensure that there are no vertices of degree two at each iteration we will obtain logarithmic collapse in the number of vertices. To ensure this property holds the authors of [13] have come up with a way of batching multiple adjacent vertices of degree two in a single iteration. If there is a path in the tree from a leaf to a degree three or more vertex where all intermediary vertices are of degree two, they should be processed in the same iteration the leaf is processed. We will call such structures leaf chains. This is in effect equivalent to contracting all vertices in the tree of degree two at every pass. This leaves only leaves and vertices of degree three or higher and ensures logarithmic collapse.

The paper [13] outlines a way of batching leaf chains when the chain is a monotone path in the tree. An issue arises when the chain is not a monotone path and some of the vertices inside it have alternating height. The we can only batch monotone subpaths and not the whole path. The more zig zags there are in the path the less monotone paths there and the more iterations we will require. This effectively serialises computation along them.

When plotted according to height such chains forms a characteristic zig-zag pattern (see [13]). We will call such chains W-Structures. They 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. We believe that theoretically it is these W-Structure that most severely hinder computation in the merge phase of the algorithm. 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 iterations 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 $O(tlog^2(t)$ work if we assume that no w-structures are present. If however there is a w-structure with more "zig zags" 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 develping some new theory about the w-structures in Chapter [] and through an empirical analysis in Chapter [].

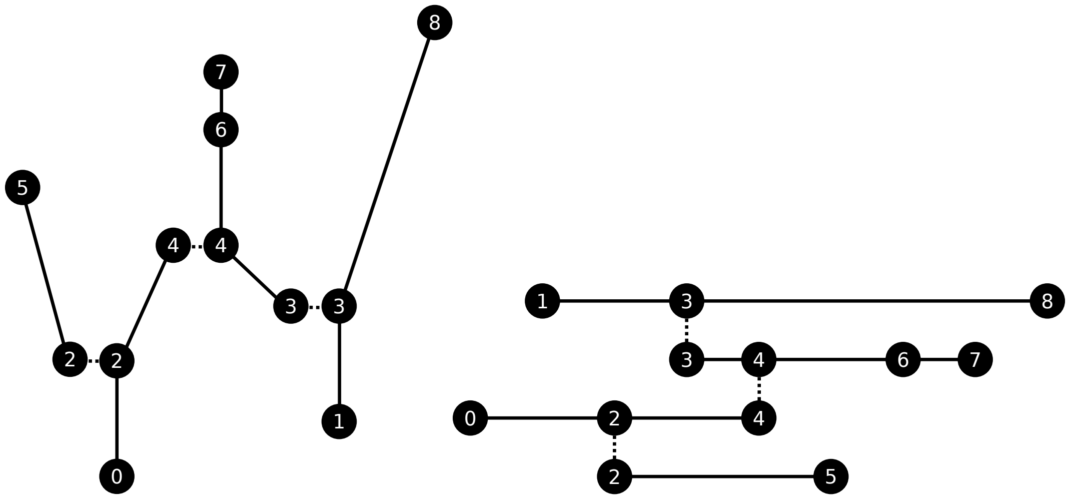
3.7 Contour Tree Simplification

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 severly limits 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.

One such technique is branch decomposition [25]. Branch decomposition involves decomposing the contour tree into a set of edge-wise disjoint monotone paths (branches) which cover all edges of the tree. A trivial branch decomposition of any tree is obtained by taking every edge to be a separate branch. Furthermore 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 it's end points and the persistence of it's children. Branches of high persistence reflect more prominent features in the tree. We apply the simplification by removing branches with low persistence that do not disconnect the tree.

The algorithm for producing the heirarchical branch decomposition of a contour tree is the following:



(a) Branch Decomposition.

(b) Heirarchical view of the branches.

Figure 3.3: Branch Decomposition of the Contour tree from Figure 3.2 [which subfigure?].

- Identify all upper leafs that connect via branches to upwards saddles.
- Identify all lower leafs that connect via branches to downwards saddles.
- Those are candidate branches. Pick the one with the lowest persistence (difference of height between the leaf and the saddle).
- Remove that branch without removing the saddle.
- Continue this process untill a single branch that connect two leaves is all that remains. That is the master branch.

Let us for example construct the heirarchical branch decomposition of the contour tree from Figure 3.2. The first two candidate branches we identify are $5-2$ and with persistence 3 and $3-8$ with persistence 5. We take the branch with lower persistence $5-2$. In the next step the candiate branches are $0-4$ and with persistence 4 and $3-8$ with persistence 5. We will take $0-4$. Afterwards the remaining candiate branches are $3-7$ with persistence 4 and $3-8$ with persistence 5. After removing $3-7$ in the final stage the only remaining branch is $1-8$. It is the master branch because it connects two leaves. The produced pairs of critical points are $(2,5),(0,4),(3,7)$ and $(1,8)$.

Branch Decomposition is a form of topological simplification whose use is limited to the contour tree. In Chapter 6 we will present a more general topological simplification method called persistent homology. Our goal will be to express branch d‘ecomposition in the framework of persistent homology and determine whether the two are equivalent.

Chapter 4

W-structures - Theory and Algorithms

We will now continue our discussion on W-Structures in a more formal setting. In this chapter we will develop theory that captures their informal description we outlined previously. We will use it to construct three general 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 correctness. Finally we will also demonstrate formal bounds on the time and space complexity of the proposed algorithms.

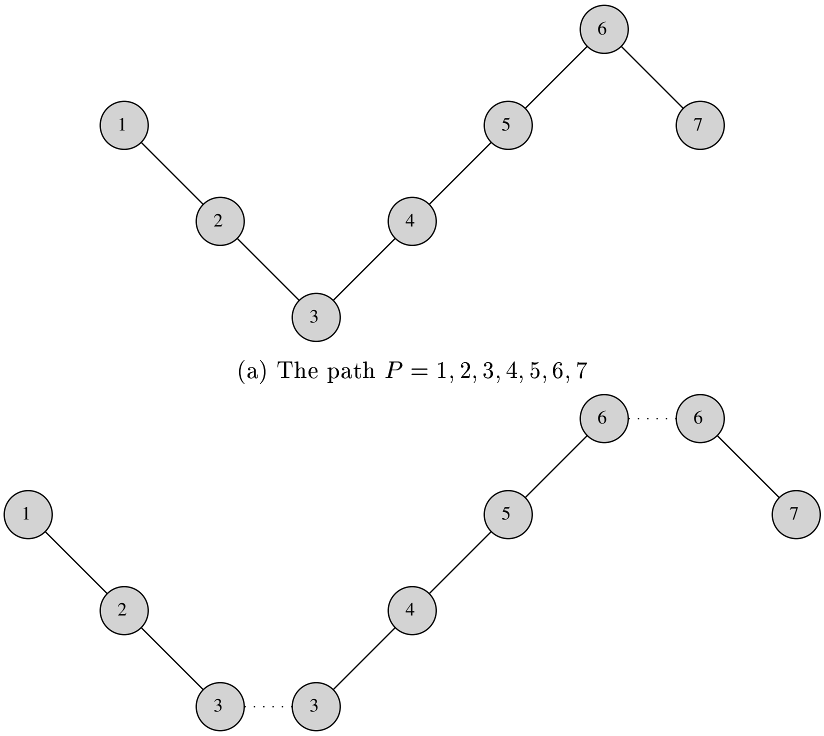
4.1 Formal Description of W-Structures

We are interested in describing paths in height trees which form the characteristic zigzag pattern we described in Chapter 3. Let us first establish some of the basic notation we shall make use of. We will consider paths to be a sequence of distinct and adjacent vertices. When dealing with paths in height trees will often 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 $v_1 \rightsquigarrow v_4$. Lastly, a subpath of a path P is a path whose vertices are also vertices of P.

The first important property of paths in height trees is their monotone path decomposition. The monotone path decompositino 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 the sequence of paths $P_1, P_2, ..., P_k$ such that $P_i \subseteq P$, $|P_i \cap P_{i+1}| = 1$ and $P_i \cup P_{i+1}$ is not a monotone path for $i \in \{1, 2, ..., k-1\}$ 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 decomposition is exactly the number of vertices in which we change direction as we traverse the path. We shall name those special vertices kinks.

A kink in a path is a vertex whose two neighbours are either both higher or both lower (Figure 4.2). Given the path $(u_1, u_2, ..., u_k)$ an inside vertex $u_i \neq u_1, u_k$ is a kink when $h(u_i) \notin \big(min(h(u_{i-1}),\ h(u_{i+1}),\ max(h(u_{i-1}),\ h(u_{i+1})\big)$. 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 $\in \big(h(u_{i-1}),\ h(u_{i+1})\big)$ 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 kinks. This is similar to how the length of a path is a metric that measures the number of edges

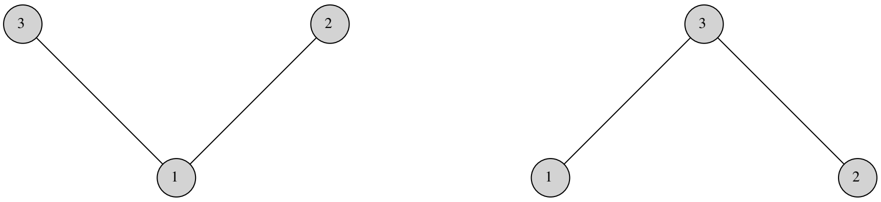


(a) The path $P = 1, 2, 3, 4, 5, 6, 7$

(b) Monotone decomposition $P_1 = 1, 2, 3$, $P_2 = 3, 4, 5, 6$ and $P_3 = 6, 7$.

Figure 4.1: A path and its monotone decomposition.

between its vertices. The notation we will adopt for the w-length and length of a path $u \rightsquigarrow u$ 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 tree. The length of a path with n vertices is $n - 1$, but at most $n - 2$ vertices can be kinks in it.



(a) Upwards kink

(b) Downwards kink

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 given contour tree. We can now put his 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 height tree and computing their w-length to find the maximum one. This can be expressed with the following optimization term

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

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 it's two neighbours in the path. The worst case running time of this algorithm is $O(d * n^2)$ where d is the diameter (longest path) of the tree and n is the number of vertices. This is however 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.

The parallel we made between the w-length and length of a path has a deeper consequences. It we were to instead ask the question of finding the longest path in a tree we would find that it is a well studied problem. Our goal now will to try to transfer that knowledge to our task at hand. We will do so by analysing how two of the most popular algorithms for computing the longest path in a tree work and whether they can be adapted to instead find the longest w-path in a tree.

After introducing these two tree diameter algorithms it is now time to demonstrate how they can be adapted to the task of finding a height tree's w-diameter. Before doing so we need to establish the two key properties that will play a crutial role in adapting the algorithms.

Definition 19. *(Subpath Property) Let $a \rightsquigarrow b$ be a path and $c \rightsquigarrow d$ its subpath. Then $w(a,b) \leq w(c,d)$.*

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 nuber of kinks in the path.

Definition 20. *(Path Decomposition Property Property) Let $a \rightsquigarrow b$ be the path $(a, u_1, u_2, ..., u_k, b)$ and u_i be an inside vertex for any $i \in \{1, 2, ..., k\}$. Then:*

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

where:

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

We claim 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 kinks are acounted 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 it will made use of a modified version of BFS that computes w-distances [see algorithm next page] 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 graph 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. This algorihm however does is not guarateed 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.

Algorithm 1 Computing the W Diameter of a Height Tree.

```
function W_BFS(T, root)
    root.d = 0
    root.$\pi$ = root
    furthest = root
    Q = $\emptyset$
    Enqueue(Q, root)
    while Q $\neq \emptyset$ do
        u = Dequeue(Q)
        if u.d > furthest.d then
            furthest = u
        for all v $\in$ T.Adj[u] do
            if v.$\pi$ == $\emptyset$ then
                v.$\pi$ = u
                if h(u) $\notin$ (h(v), h(u.$\pi$)) then
                    v.d = u.d + 1
                else
                    v.d = u.d
                Enqueue(Q, v)
    Return furthest
function CALCULATE_W_DIAMETER(T)
    s = <any vertex>
    u = W_BFS(T, s)
    v = W_BFS(T, u)
    return v.d
```

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 in the tree whose w-length is at least that of the w-diameter of the tree 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:

$$w(s,u) \geq w(s,t), \forall t \in V(T) \tag{4.1}$$

$$w(u,v) \geq w(u,t), \forall t \in V(T). \tag{4.2}$$

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

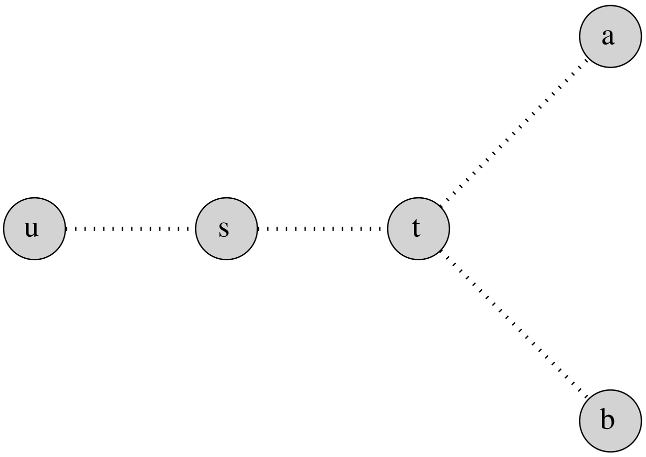


Figure 4.3: Relative position of vertices in Case 1.1

$$w(a,b) \geq w(c,d), \forall c,d \in V(T) \tag{4.3}$$

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 $w(a,b)$. To this end let t be the first vertex in the path between a and b that the first BFS starting at s discovers. We can infer that t cannot be a or b unless s is equal to a or b.

The proof can then 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.

In this case $s \rightsquigarrow u$ is a subpath of $t \rightsquigarrow u$, which in turn 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 $w(t,u) \geq w(a,t)$ as $s \rightsquigarrow a$ is a subpath of $t \rightsquigarrow a$.

Now via path decomposition of $a \rightsquigarrow b$ and $u \rightsquigarrow b$ at t have that:

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

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

Where $x, y \in \{0,1\}$ 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:

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

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

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

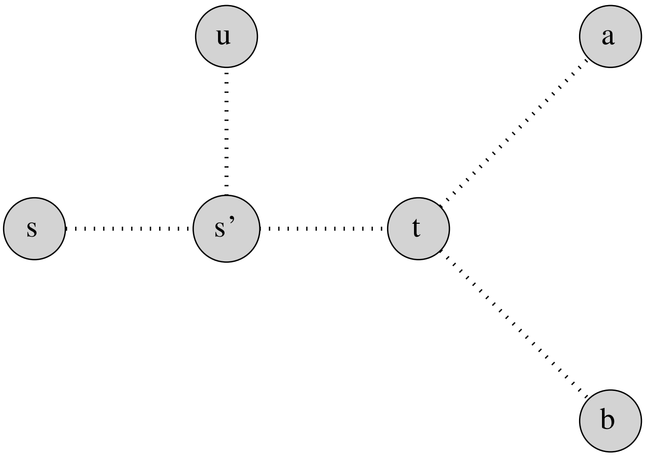


Figure 4.4: Relative position of vertices in Case 1.2

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

But as $w(u,v) \geq w(u,b)$ (by equation 4.2) we obtain that: Sha

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

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 $x=1$. 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.

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 be able to produce s 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 $s \rightsquigarrow t$. We know that $w(t,u) \geq w(s',u)$ (as a subpath) and 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 $x, y \in \{0,1\}$ indicate whether s' is a kink in the corresponding path as before. By equation 4.1 we know that $w(s,u) \geq w(s,a)$ and therefore:

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

Since s' lies on the path from t to u we have that $w(t,u) \geq w(s',u)$ by the subpath property.

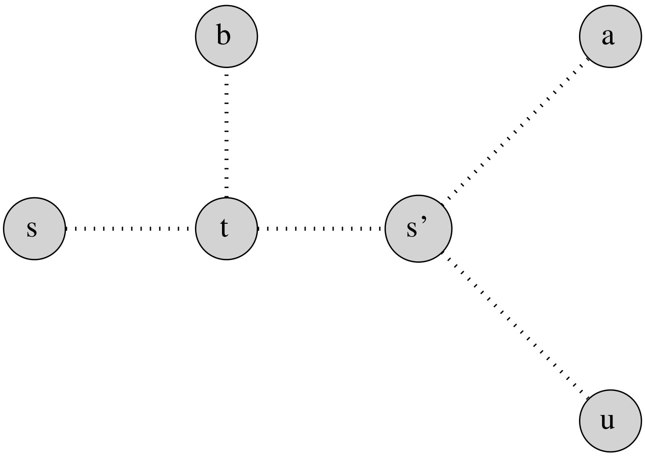


Figure 4.5: Relative position of vertices in Case 2 (t could be equal to s').

We can use this to conclude the following:

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

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:

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

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

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

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

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

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

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:

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

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

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:

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

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

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

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

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

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

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

Based on these cases we can 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 maximum w-path.

□

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 $O(|V|)$.*

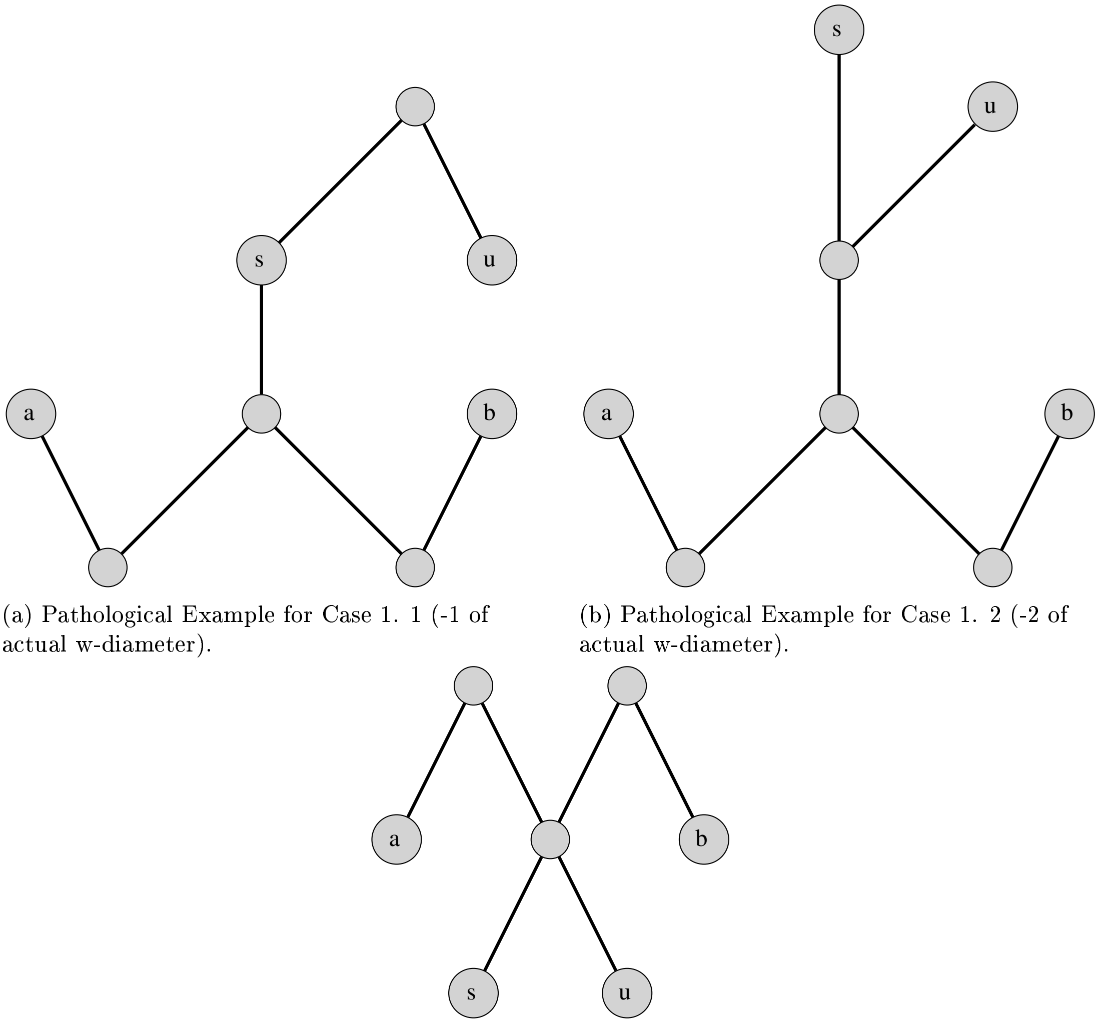
Proof. The modified BFS function has the same time complexity as BFS. All we have added to the standard BFS is an "*if, then, else*" statement. The time complexity of BFS is $O(|V| + |E|)$, but in a tree $|E| = |V| - 1$, so the overall complexity is $O(2|V| - 1) = O(|V|)$. Running the modified BFS function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

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

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

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



(a) Pathological Example for Case 1. 1 (-1 of actual w-diameter).

(b) Pathological Example for Case 1. 2 (-2 of actual w-diameter).

(c) Pathological Example for Case 2 (-2 of actual 2-diameter).

Figure 4.6: Branch Decomposition of a Contour tree.

corresponds to one of the cases in Lemma []. In all examples the initial vertex is taked to be s. After running the algorithm we can see that the vertex outputted by the first BFS function would be u after which the longest path would be outputted as $u \rightsquigarrow a$ or $u \rightsquigarrow b$. We can see that that in all figures $w(u, a) = w(u, b) = 1$ or 2, but $w(a, b) = 3$.

Even if we adapt the algorithm so that it finds the vertex with that is farthest in terms of both w-length and length we will still be able to make simillar pathological case examples. We just need to augment the height trees by making u further away from s that a and b but keeping the same relative w-length of all paths.

4.2.2 Attempts at resolving the accuracy of 2xBFS

In this section we adapted the first of the tree diameter algorithms to obtain a w-diameter algorithm with what we claim is reasonable accurate. That accuracy is supported by Lemma [], but we have not way of knowing whether the w-path obtained through 2xBFS is optimal of not. Here we will present two possible ways of improving the accuracy of the output of the 2xBFS algorithm and explain why were not able improve upon the theoretical bound of Lemma [].

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 nth 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. Obverse how in Figure [] all iterations of BFS go from the vertex u to the vertex v and then from v to u and so on.

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 more reliable for if we run the algorithm from all vertices in the tree we will obtain the actual w-diameter. The ussue in doing is that the time complexity will become quadratic and we would be no better off that with the exhaustive brute force approach. If however we run the algorithm for some subset of the vertices in the tree we lose all guaratees 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 that we outlined previously. We will use the same optimisation strategy i.e. dynamic programming by making two key changes. Instead of the function $h(u)$ that computes 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. We will remane the function that stores the value of the optimal solution for subroblems 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.

Similarly to the modification of the BFS based algorithm all additional difficulties stem from the difference in the properties of length and w-legnth 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 $h(s) = \max\limits_{u \in N(s)} \big(h(u)\big) + 1$. 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 $w(u) = \max\limits_{v \in N(s)} (w(v))$. 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 it's children and $L_u = \{u_1, u_2, ..., u_k\}$ 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: $w(s) = \max\limits_{u \in N(s)} \{h(u) + \max\limits_{v \in L_u}(w_{s \rightsquigarrow v}(u))\}$. 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 the paths that end 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. 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 longest paths and at 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 combinig paths extends 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 a rooted tree 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:

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

We must pay attention to one special case. This is when the root of a rooted height tree has exacly one child. We cannot make a path combination in that case so we will assume that the formula will compute the w-height of the tree instead.

We now claim that 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. Combining what we have shows so far we obtain the following expression for the optimal solution:

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

Let us now 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 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. Lasty we have that $w_{u' \rightsquigarrow v'}(s) = w_{u \rightsquigarrow v}(s)$ because $u \rightsquigarrow v$ is a subpath of $u' \rightsquigarrow v'$. 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 Equation []. By observing the two carefully we can infer that

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

and

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

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:

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

The w-path combination we have presented produces a valid path in the tree. 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 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-dimater algorithms is shown in Algorithm [].

Having proven that the algorithm correctly computes the desired w-diameter we will now provide formal bounds on the time and space complexity of the proposed solution. We can summarise the time complexity in the following formula:

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

where we use $d(u)$ for the degree of a vetex. 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 wil begin by showing that:

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

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 is were possible. Let T be a height tree with root s and 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. 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 bellow 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 $x \in Z^+$ $x^2 \geq x$. Let us now work our way to bounding the term 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 edge we obtain that:

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

The key to tranforming this inequality is to notice is that if we expand the summation $\sum_{uv \in E(T)} d(u) + d(v)$ then every term $d(u)$ will be present exactly $d(u)$ times. One time for each one of it's 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've obtained so far:

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

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

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

The worst case running time we obtained for the DP algorithm is quadratic. The bound is tight by Equation []. This does not bode well for us especially since the brute force approach is quadratic as well. We do however believe that this worst case is very 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 abstrain 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. In the next section we will expand on the details of the implementations and in the final chapter of the dissertation we will present the results from the running time comparison of the implementations.

Algorithm 2 Computing the W Diameter of a Height Tree.

```
FunctionW_DFS(T, s)
// Base Case
if |T.$Adj$[s]| == 1 AND s.$\pi \neq$ s then
    s.W = 0
    s.w = 0
    return
// DFS Visit
for all u $\in$ T.$Adj$[s] do
    if u.$\pi$ == $\emptyset$ then
        u.$\pi$ = s
        W_DFS(T, u)

// After all neighbours are visited
// Calculate w-height of s
for all u $\in$ T.$Adj$[s] do
    if L[u] == $\emptyset$ then
        H[s] = max(H[s], H[u]);
    else
        for all v $\in$ L[u] do
            H[s] = max(H[s], H[u] + $w_{v,s}(u)$);
// Find all children that contribute to the a w-height path
for all u $\in$ T.$Adj$[s] do
    if L[u] == $\emptyset$ AND H[s] == H[u] then
        L[s] = L[s] $\cup$ u
    else
        for all v $\in$ L[u] do
            if H[s] = H[u] + $w_{v,s}(u)$ then
                L[s] = L[s] $\cup$ u
// Find the maximum path combination
maxCombine = 0
for all u $\in$ T.$Adj$[s] do
    for all v $\in$ T.$Adj$[s] do
        if v == u then
            continue
        temp = H[u] + H[v]
        if L[u] $\neq \emptyset$ then
            for all t $\in$ L[u] do
                if $w_{t,s}(u)$ == 1 then
                    temp = temp + 1
                    break
        if L[v] $\neq \emptyset$ then
            for all t $\in$ L[v] do
                if $w_{t,s}(v)$ == 1 then
                    temp = temp + 1
                    break
        if $w_{u,v}(s)$ == 1 then
            temp = temp + 1
        maxCombine = max(maxCombine, temp)
     // If there is exacly one child maxCombine will not have been
    maxCombine = max(H[s], maxCombine);
```

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

```
// Find maximum subproblem solution
for all $u \in T.Adj[s]$ do
    O[s] = max(O[s], O[u])
// Take the bigger of the two
O[s] = max(O[s], maxCombine)
function CALCULATE_W_DIAMETER(T)
    s = <any vertex>
    $s.\pi = s$
    W_DFS(T, s)
    return s.W
```

Chapter 5

Homology

In this chapter we will shift our attention back to algebraic topology and more specifically the field of Homology. We will use Homology to analyse the connectivity, number of the holes and voids in a simplical complex. We are putting all this work in Homology because it is a prerequisite to one of the leading tools in topological data analysis - Persistent Homology. In the next chapter we will analyse some of the similarities between persistent homology computations and contour tree computations.

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 [10]. For example with homology we can recognize that there is a hole in the torus and a volume enclosed in the sphere. The theory of Homology comes in two flavours - **simplical** and **singular**. Simplicial homology is geared towards analysing simplical complexes and singular homology is the appropriate generalisation for arbitrary topological spaces. In this dissertation we restrict attention on singular homology because we are primarily interested in the computational aspect of homology.

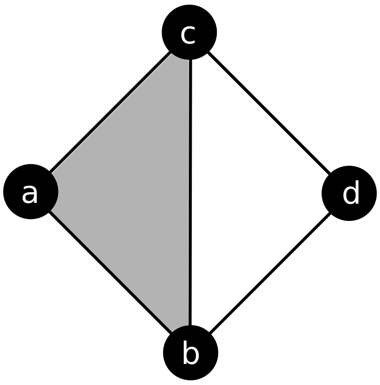


Figure 5.1: An Example Simplical Complex

Homology is built around the interplay between two key concepts of **cycles** and **boundaries**. Let us consider the simplical complex depicted on Figure 5.1 as an example. It consists of four vertices $\{a, b, c, d\}$, five edges $\{ab, bc, ca, db, cb\}$ and one face $\{abc\}$. Let us first explain what a boundary is. The boundary of a simplex consists of its codimension-1 faces. For example the boundary of the 1-dim simplex ab consists of the 0-dim simplices a and b. The boundary of the 2-dim simplex abc consists of the 1-dim simplices ab, ac and cb. A cycle on the other hand consists of the simplices that form the boundary of a simplex that is of one dimension higher (regardless of whether that one dimension simplex is in the complex). In our example we can observe that the edges ab, bc, ca and bd, dc, cb form a 1-dim cycle because they are the boundary of the-dimensional simlices abc and bdc. The first simplex abc is in the complex while bdc is not. The definition of one dimensional cycles is in line with the graph theoretic definition. The first and last vertex of the paths formed by those edges are the same. A more geometric way to put it is that the edges enclose an 2-dim area of space. To expand this definition to higher dimensional cycles picture the faces of the tetrahedron. They would form a 2-cycle as they completely enclose a 3-dim volume. In general an n-cycle consists of simplices that are the boundary of a n+1-dim simplex

The interplay between between cycles and boundaries is in asking the 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 space. Cycles which are the boundary of a higher dimensional simplex can be disregarded because the void they introduces is filled by that higher dimensional simplex. Coming back to our example the cycle ab, bc, ac is the boundary of abc, but the cycle db, cd, bc it not the boundary of another simplex in the simplical complex. The cycle db, cd, bc represents a 2-dim hole in the simplical complex. Finally note that the cycle ab, bd, dc, ac is in a sense equivalent to the cycle db, cd, bc because both describe the same 2-dim hole in the complex - namerly the missing 2-dim simplex bdc.

Notice also that the paths formed by the edges bc, ca, ab and ac, ab, bc represent the same cycle. The only difference is which vertex is the starting and ending point. We would like to disregard the choice of starting point and order of edges completely because for example the paths bc, ac, ab and ac, ab, bc represent the same structure in the simplical complex. Enter 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 to 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. To begin with, we will operate 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 use abstract vector spaces over the field of coefficients $\mathbb{Z}_2$ as the building blocks of our study of Homology.

Let X be a simplicial complex. An n-chain of X is a formal sum of n-simplices of X [9]. 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 $(ab + bc) + (ab + cd + db) = 2ab + bc + cd + db = bc + cd + bd$ 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 the vector spaces where the vectors are all possible n-chains of X and the coeficients 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 $\partial_n : C_n(X) \to C_{n-1}(X)$.

What we have defined as the chain complex of X is no more that a collection of vector spaces together with linear maps between. When n is smaller than zero bigger than the dimension of X then the approriate vector space is the trivial, consisting only of the zero element. We can visualise the chain complex of X with the so called quiver representation. For our example simplical 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 simplical complex X the full chain complex would be:

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

where we can extend both right and lefthand 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 expand on how the groups of n-chains and the boundary maps of a simplical complex are constructed explicitly. In our working example $C_0(X)$ is the vector space that is spanned by the vertices $\{a, b, c, d\}$ of the complex. We write this as $C_0(X) = span(\{a, b, c, d\})$. 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 a vector, 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 $C_1(X) = span(\{ab, bc, ca, cd, bd\})$. As we pointed out earlier the cycle that consists of the edges bc, cd, db is represented by the sum or linear combination $bc + dc + bd = 0ab + 1bc + 0ca + 1cd + 1bd$ and has coordinates $(0, 1, 0, 1, 1)$ in $C_1(X)$ with respect to the basis we have chosen. We may of course work in any basis we like. For example $C_0(X) = span(\{a + b, b, c, c + d\})$ 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 $a + b + c + d$ will have coordinates $(1, 0, 01)$.

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 consisting of the simplices of $C_{n-1}(X)$ that are codimension-1 faces of σ. If σ is the convex combination of the vertices $[v_0, v_1, ..., v_n]$ then we define it's boundary as

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

where the hat on top of v_i signifies that we omit it in the convex combination. From this definition we can extend ∂ linearly by allowing it commute with vector addition and scalar multiplication like so:

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

Going back to our working example let $\sigma = ab + bc + ca$ be an n-chain in $C_1(X)$. Then $\partial(ab + bc + ca) = \partial(ab) + \partial(bc) + \partial(ca) = a + b + b + c + c + a = 2a + 2b + 2c = 0$. 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 it's boundary is zero. This is no concidence. The defining feature of a cycles is that they have no boundary. In general the n-cycles in $C_n(X)$ are exactly the n-chains that go to zero under the boundary map. The set of all vector that go to zero under a linear map is known as the kernel of the linear map. The kernel of the boundary map $\partial_n : C_n(X) \to C_{n-1}(X)$ is denoted as:

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

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:

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

Now that we have the means of describing the cycles and boundaries the only thing that we are missing is to partition the cycles into groups of cycles that differ from each other only by their boundary. We want of way of making precise the notion that the cycles ab, bd, dc, ac and db, cd, bc in Example [] are equivalent because they both represent the missing simplex dbc. To do so we must first understand how Z_n and B_n are related through the fundamental Lemma of Homology.

Lemma 7. *Fundamental Lemma of Homology.* $(\partial_{n-1} \circ \partial_n)(\sigma) = 0,$ *for every* $\sigma \in C_n(X)$.

Proof. We will only sketch the intuitive outline of the proof and refer the reader to [17] for a more complete 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 sigma. Therefore they will cancel out in the second boundary operation $\partial_{n-1}\partial_n(\sigma)$. □

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

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 $(\partial_{n-1} \circ \partial_n)(\sigma) \neq 0$. By the Fundamental Lemma of Homology this is not possible. □

We have found that $B_n \subseteq Z_n$, but we can make an even stronger statement. From linear algebra [1] we know that the kernel and image of a linear function 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 bounday in B_n we can take the quotient of the two spaces. This quotient is in the heart of Homology!

Definition 21. *The n-th homology group of a chain map is the quotient* $H_n(X) = Z_p/B_p = ker(\partial_{n+1})/im(\partial_n)$.

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 [1]. 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 [1]. Therefore $H_n(X)$ is a vector space and $dim(H_n(X)) = dim(ker(\partial_{n+1})) - dim(im(\partial_n))$. The elements of $H_n(X)$ are called homology classes. For a cycle $\sigma \in Z_p$ we done it's 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 $[ab + bd + dc + ac] = [db + cd + bc]$. Both cycles are representatives of the same homology class.

The dimensions of the homology groups are a summary of the topological information about the connectivity of the n-dimensional simplicies 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
- Betti one or $\beta_1 = dim(H_1)$ is the number one dimentional holes in a space or holes.
- Betti two or $\beta_2 = dim(H_2)$ is the number two dimentional holes in a space or voids.

The higher Betti numbers represent the number of higher dimensional voids. In a simplicial complex of finite dimension the Betting numbers from a point onwards to all be zero. This of course means that the according homology group are the zero dimensional vector space.

Homology computations are far too envolved and lenghty to be given as examples here. We refer the eager and interested reader to [].

5.2 Reduced and Relative Homology

There are two extensions of homology we need to discuss so that we may to be able to fully harness the power of persistent homology in the following chapter. 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 simplical complex that consists of a single vertex. All of its homology groups except for the H_0 are trivial. It is convenient in many application to force H_0 behave like the rest of the homology group. More specifically, in our example of a single vertex we would like for it's 0th homology group it to be trivial. Consequently we would also like path-connected simplical complexes will have trivial 0th homology. The geometrical interpretation of this extension is the reduced 0th homology classes represent the number of voids that separate path connected components and not the path connected components themselves.

In order to accomplish this we will augment the chain complex of simplical complex X with one additional group $\mathbb{Z}_2$ and one linear map $\epsilon : C_0(X) \to \mathbb{Z}_2$. The resulting chain complex is:

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

In this augmented chain the function $\epsilon : C_0(X) \to \mathbb{Z}_2$ is defined as $\epsilon\big(\sum_i n_i\sigma_i\big) = \sum_i n_i$. The value of ϵ is equal to the parity of the number of simplicies in the chain. We will define the reduced homology as the homology of the augmented chain complex or $\widetilde{H}_n(X)$. From [17] we have that $\widetilde{H}_n(X) = H_n(X)$ for $n > 0$ and $\widetilde{H}_0(X) \bigoplus \mathbb{Z}_2 = H_0(X)$.

Another crucial concept is that of relative homology. Relative homology aims to simplify the homology of a simplical 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 quotients 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 bounday map takes a relative class from $[\sigma] \in C_n(X, A)$ to a relative class $[\partial_n(\sigma)] \in C_n(X, A)$. By taking the relative chain groups together with the relative chain maps we obtain the relative chain complex.

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

We will define the relative homology groups of the relative chain complex as $H_n(X, A) = ker(\partial_n)/im(\partial_{n-1})$ where we substitute ∂_n and ∂_{n-1} to be the relative boundary groups. 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 [17].

- A relative chain α is a relative cycle when it's boundary $\partial_n(\alpha)$ is in $C_n(A)$.
- A relative cycle α is trivial in the homology when it's the sum of a boundary $\partial_n(\beta)$ 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 [12]. In fact they are equal when we quotient by a single vertex of X. Let p be a 0-simplex of X then $\widetilde{H}_n(X) = H_n(X, p)$. The reason for this is that the 0th homology class p becomes trivial in the homology relative to p.

The relative homology classes are a purely algebraic construction, but for simplical complexes there is an appropriate geometric intuition that goes along with them. It is expressed through the following theorem [9].

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 [12] is that if A is a subcomplex of X then $H_n(X, A) \simeq H_n(X/A, A/A) \simeq \widetilde{H}_n(X/A)$ where A/A is a single point in X/A. This will allows us to leverage our geometric intuition about quotient spaces to compute homology groups. 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 simplical complex we will use the Excision Theorem to compute the dimension of $\widetilde{H}_0(X/A)$ 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 final 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 22. *Let X be a simplicial complex and A be a subcomplex of X. A function $i : A \to X$ 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 maps is the identity map. Note that an inclusion maps is a special case of a simplicial map [18]. A simplicial map between two simplical complexes takes simplicies from one to simplicies of the other. An inclusion map fits that cireria by simply mapping all simplicies to themselves. Furthermore inclusion maps will allows us to obtain maps between the chain groups of A and X.

Definition 23. *Let X be a simplicial complex and A be a subcomplex of X and $i : A \to X$ be an inclusion map. Then i induces an inclusion map $i_{\#} : C_n(A) \to C_n(X)$ for all $n \in \mathbb{Z}$.*

In order to define $i_{\#}$ we just have to extend i linearly 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 maps because every n-chain in $C_n(A)$ is also an n-chain of $C_n(X)$ and i maps simplicies to themselves. Upon obtaining inclusion maps between the chain complexes of A and X we can take a step further and induce a linear map between the homology groups of A and X.

Definition 24. *Let X be a simplicial complex and A be a subcomplex of X and $i_{\#} : C_n(A) \to C_n(X)$ be an inclusion map. Then $i_{\#}$ induces an linear map $i_* : H_n(X) \to H_n(Y)$ such that $i_*([\sigma]) = [i_{\#}(\sigma)]$ for all $n \in \mathbb{Z}$.*

Where $[\sigma]$ is the homology group of a an n-chain σ in $C_n(A)$ and $[i_{\#}(\sigma)]$ is the homology group of a an n-chain $i_{\#}(\sigma) = \sigma$ in $C_n(X)$. A crutial 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 $H_n(A)$ if X contain a simplex whos boundary is σ, but A does not not.

The last thing we will do it to expand the definitions we've made for far to relative homology groups.

Definition 25. *Let X be a simplicial complex. Let A and B be two subcomplexes X such that $A \subseteq B$. Then the identity map $i : X \to X$ induces a linear map $i_* : H_n(X, A) \to H_n(X, B)$ for all $n \in \mathbb{Z}$.*

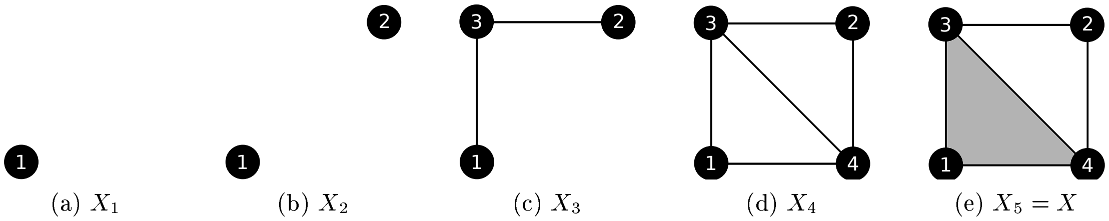
This will call this inclusion of pairs. The pairs are (X, A) and (X, B). To see why this holds we refer the reader to [17]. One just has to keep in mind that the identity map is a well defined continous map between topological pairs such as (X, A) and (X, B).

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 signicance of topological features. The primary motivation for introducing persistent homology in the first place was the practical need to cope with noise in data [9]. 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 [10] as a tool for automated topological simplification. The building blocks of persistent homoogy are sequences of nested simplicial complexes called filtrations. A filtration of a simplicial complex X is a one parameter family of simplicial complexes $\{X_t\}_{t\in\{0,...,n\}}$ where $X_i \subseteq X_j$ whenever $i \leq j$ and $X = X_n$. If we arange the consequtive X_i in a linear sequence we obtain the following:

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



(a) X_1 (b) X_2 (c) X_3 (d) X_4 (e) $X_5 = X$

Figure 5.2: Example of an 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 simplical 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 $i : X_k \to X_{k+1}$ for $k \in \{0, 1, ..., n-1\}$. Using those inclusion maps we can build the following chain of simplicial complexes.

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

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:

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

for all $p \in \{0, 1, 2, ...\}$. 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 [12]:

- 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.

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. In such a situation we must make a choice on which one of the classes dies and which of the persists. It does not matter what we choose as long as we remember the choise in the future. In order to be consistent in choosing we will apply the Elder Rule [9]. According to the elder rule the class whose birth time is smaller will persisnt and the other one will die. In the case of multiple classes we merging all will die except the oldest.

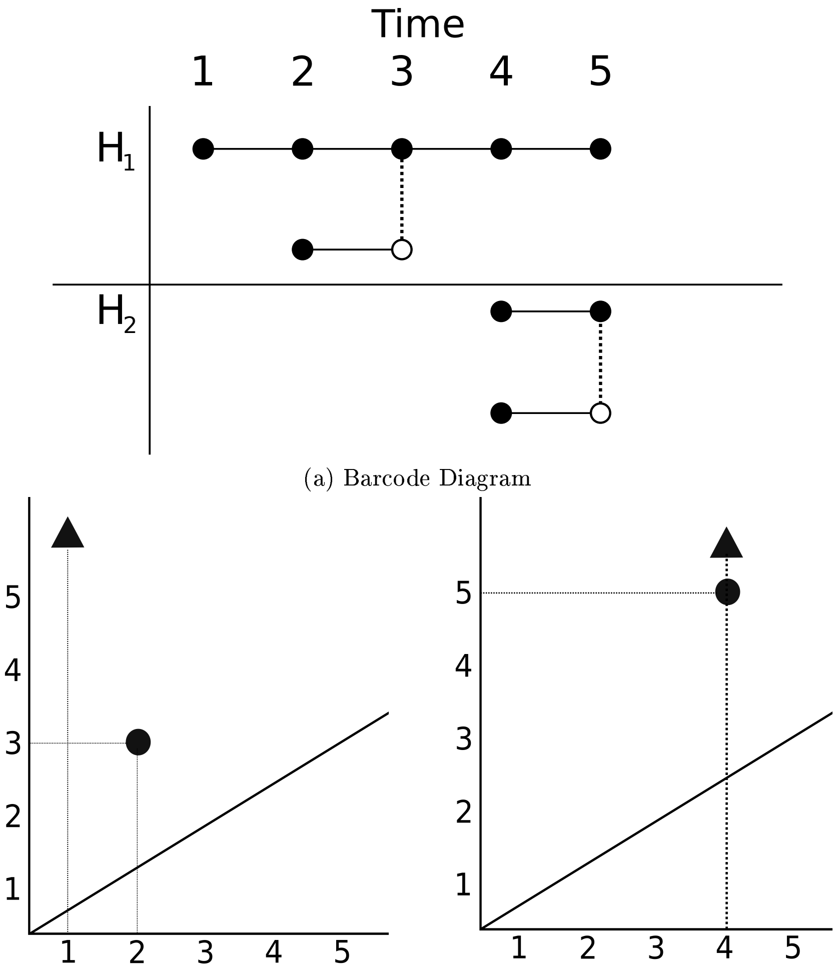
Using the language of birth and death we can define the persistence of a homology class. Let α be a homology class that is born in X_i and dies in X_j. We call the difference $j-i$ the persistence of α. Some classes however do not have a defined death time. These are the classes of the final complex 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 formed as (t, ∞) where their birth time is t and we set their death time to ∞ due ot the lack of one. We visualise the persistence pairs by plotting as points in the plane. This is called a persistence diagram.

Another way to visualy persistent homology is via a barcode diagram [11]. 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 we are in and the vertical axis coresponds 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 our working example of a filtration is on Figure **??**.

afasd

You can see in the digram that one 0th homology class is born at time 1 and another one at time 2. In time 3 the class born at time 1 merges with the class that is born at time 1 because it is the younger class. In time 4 on the other hand two cycles or 1st homology classes are born.



(a) Barcode Diagram

(b) Pairs in H_0

(c) Pairs in H_1

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

One of them dies at time 5 and the other one pesists. These events corresponds to the persistent homology pairs $(2,3),(4,5),(1,\infty)$ and $(4,\infty)$.

Now let us apply the general theory of persistent homology to a more familiar domain. Let M be a traingulation of a bounded area in $\mathbb{R}^2$ and let $f : M \to \mathbb{R}$ be our familiar linear interpolat. We would like to obtain a filtration of M that we can analyse with persistent homology. The filtration that is proposed in the literature [9] is of the sublevel sets M. As we have already shown in Chapter 2 Morse functions have finitely many critical points, changes in the topology of the sublevel sets happen only are critical points and critical points are at the vertices of M. Therefore we need only consider the sublevel sets at the critical values. Let $c_1 < c_2 < ... < c_n$ 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 intirely in M_{c_i} in order to obtain a simplicial complex. This lets us obtain a filtration of the sublevel sets:

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

From this filtration we can produce the following chain of homology groups:

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

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.

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

We are now able to compute both the contour tree and its branch decomposition and persistent homology. Our motivation behind doing so has been sparked by a quote from the original paper that introduced branch decomposition of contour trees [25]. Branch decomposition results in a set of branches or equivalently pairings of critiral points of the contour tree. In the branch decomposition paper the importance of branches is based on their persistence as defined in the original paper that introduced persistent homology [10].

This idea of the similarity between the two however has not been explained in detal nor explored further in the paper or in subsequent publications. We would like to explore it further. We would like to test whether the pairings produced from branch decomposition are the same as the pairings produced by persistent homology. We will do so by computing both on the same data sets and comparing the results to see how and why they differ. Before doing so however we must address an inconsistency between the two. The branch decomposition of the contour tree pair all critical points. Some critical points in persistent homology however are not "properly" paired. These are the ones that corespond to essential homology classes which have infinite persistence. To address this issue we will introduce an extension to persistent homology that "properly" pairs essential homology classes by assinging them with finite persistence.

5.5 Extended Persistence

We have seen from the definition and computations of persistent homology that not all critical points are paired. Those that give birth to the essential classes will not be paired because they are never destroyed pass the final simplex in the filtration. This leads to incompleteness in the persistence pairings which we would to remedy. Our goal in extending persistence it to devise a way to pair the essential homology classes with the remaining unpaired critical points.

Let M be a traingulation of a bounded area in $\mathbb{R}^2$ and let $f : M \to \mathbb{R}$ be a linear interpolat. The main idea behind extended persistence is to follow the ascending filtration of persistent homology with a descending pass where once we reach a class that is homologous to a essential class in the ascending filtration we consider it to be destroyed and thus paired. Extended persistence consists of two consecutive sequences. The first sequence is the made up of absolute homology groups going up:

$$0 = H_n(M_{c_1}) \to H_n(M_{c_2}) \to ... \to H_n(M_{c_{n-1}}) \to H_n(M_{c_n}) = H_n(M)$$

Just like in orginary persistence the linear maps between consecutive absolute homology gruops 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:

$$H_n(M) = H_n(M, M^{c_n}) \to H_n(M, M^{c_{n-1}}) \to ... \to H_n(M, M^{c_2}) \to H_n(M, M^{c_1}) = 0.$$

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 [] 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 $H_n(M_{c_n}) = H_n(M) = H_n(M, M^{c_n})$ we obtain the following single sequence:

$$0 = H_n(M_{c_1}) \to ... \to H_n(M_{c_n}) = H_n(M, M^{c_n}) \to ... \to H_n(M, M^{c_1}) = 0.$$

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. A good way of picturing extendex persistence if through excision. Take a look at [30] for more details examples of this.

Chapter 6

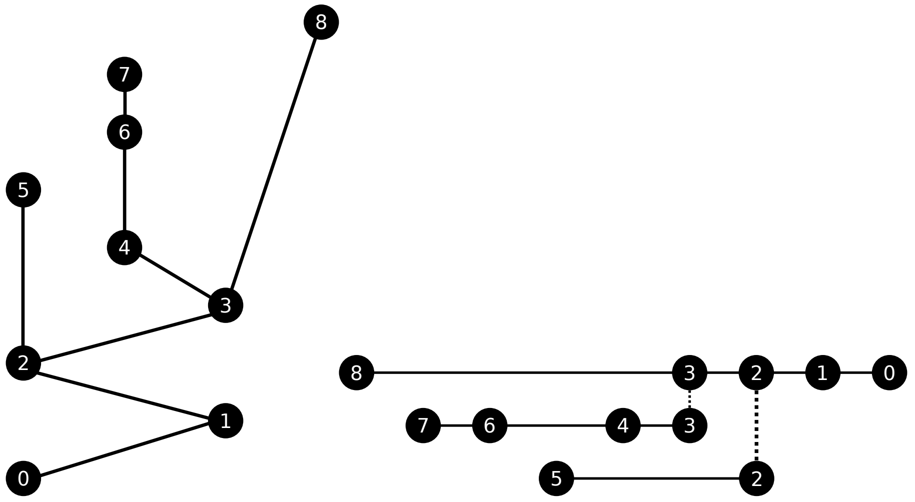
Extended Persistence and Branch Decomposition

We can already see from this premise that the idea behind persistent homology is akin to that of contour tree simplification. In this chapter we will introduce persistent homology and in the next we will apply it to contour trees and explore its relation to contour tree splification via branch decomposition. In this chapter we will explore the relation between the pairs of critical points obained via branch decomposition and via extended persistence. We will introduce the historical relation between the two in the literature and the proceede to explore it with an example data set.

6.1 Persistence of Branches

In January of 2004 Pascucci at. al publish the original paper that introduces branch decomposition []. The authors of the paper clearly state that there is a similarity between their new method and the more general and recently established method of persistent homology (as it was originallly introduced in []). They do not go further in describing in detail how the persistence of branches is derived formally withing the framework of persistent homology. The relation between the two is not fully explored in the paper nor in subsequent publication. Describing it formally in a complete and rigrous 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 persistent homology of the zeroth homology?

We can immediately see 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 extened persistence, but extended persistence was firmly established in a paper by Herbert et. al published five years later in 2009. It is however not the case the the concept of pairing the essential classes was not unheard of at the time of publishing of []. A previous paper [] by Herbert et. al published in early 2004 outlines a method of pairing the essential homology classes of two-manifolds. It is completely reasonable to assume that Pascucci et. al were aware of this recent devlopment. Since the method introduced in [] is a special case of what was later defined as extended peristence we will extend our comparison to the pairs of critical points obtain by extended persistence. Are the pairs of critical points produced by branch decomposition the same as the pairs produced by extended persistent of the zeroth homology?



(a) Join Tree (b) Branch Decomposition

Figure 6.1: Branch Decomposition of the join tree.

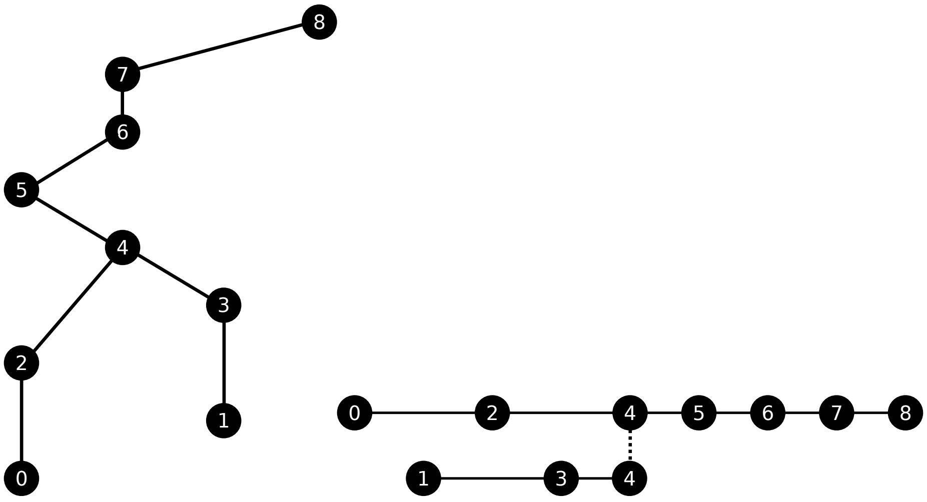
6.2 Persistence Pairs vs Branch Decomposition Pairs

In this section we will present an example which demonstrates that at least one pair of critical points is different in branch decomposion and in extended persistence. This example will be based on Figure 3.1 and our familir 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 [] then branches obtained via branch decomposition correspond to valid topological cancelation in the simplicial mesh as well []. Therefore we can compare it with extended persistence of the simplicial mesh. For completeness 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 d) 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.

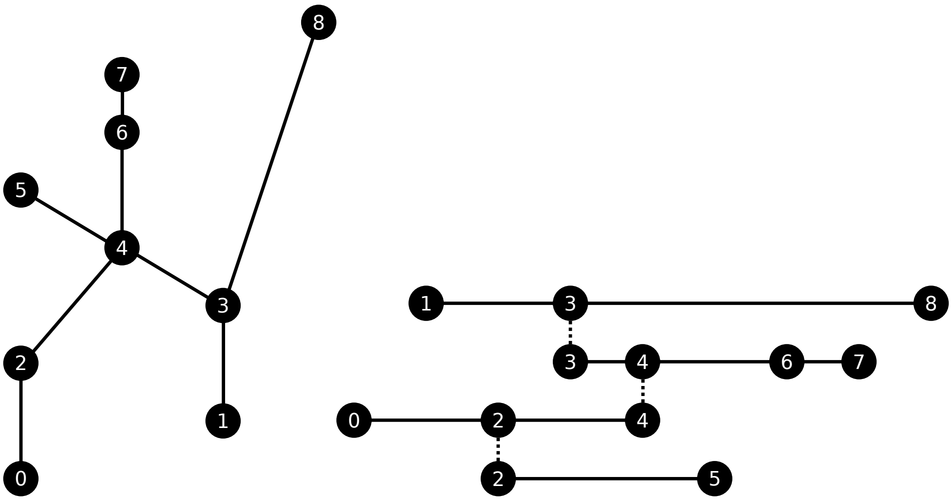
We will examine the join, split and contour trees first. We already computed the branch decomposition of the contour tree in chapter []. For the join tree the candidate branches are $5-2$, $7-3$ and $8-3$. We remove them in order of persistence. First we remove $5-2$, then $7-3$ and afterwards we remove the root branch $8-0$ 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 $0-8$.

Next we will present the ascending filtration of the simplicial mesh and its extened persistence. The ascending filtration of the mesh is obtained by adding all the vertices one at a time (and



(a) Split Tree (b) Branch Decomposition

Figure 6.2: Branch Decomposition of the split tree.



(a) Contour Tree (b) Branch Decomposition

Figure 6.3: Branch Decomposition of the contour tree.

edges between them) in order of their height. We can obtain the persistence pairs through inspection because connected components correspond to persistence homology classes. Obsertve 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 0. Therefore the first persistence pair is $(1, 4)$. The following complexes of the filtration consisnt 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.

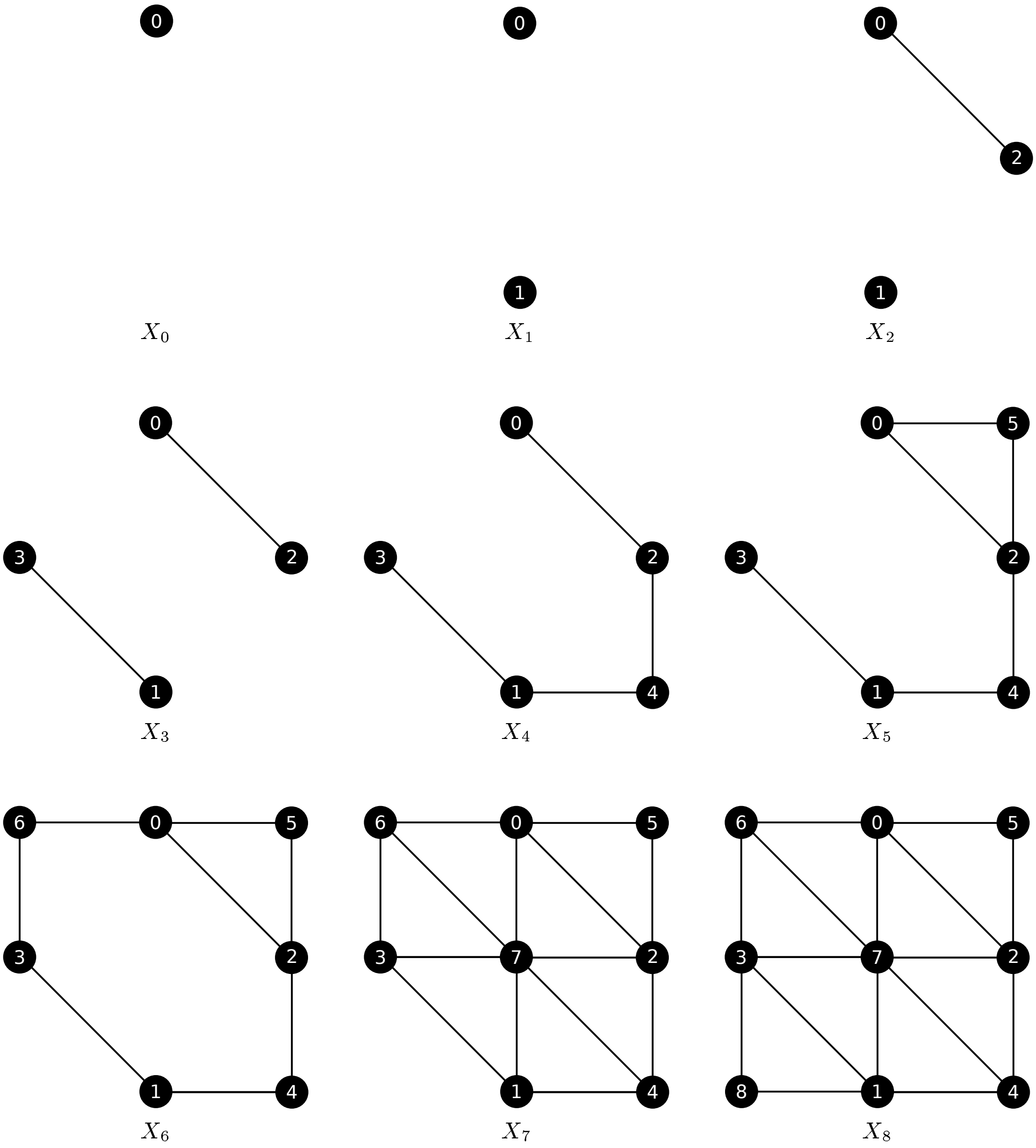


Figure 6.4: Ascending Filtration of the Dataset.

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 he 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 $H_0(X, \{8\}) = \widetilde{H}_0(X/\{8\})$. By [] we 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 $dim(\widetilde{H}_0(X/\{8\})) = dim(H_0(X/\{8\})) - 1 = 0$. A zero dimensional vector space is the trivial vector space consisting only of the zero vector. The map $i_* : H_0(X_8) \to H_0(X, X^8)$ is the zero map because $H_0(X, X^8) = H_0(X, \{8\}) = \widetilde{H}_0(X/\{8\}) = 0$. 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 $(0, 8)$.

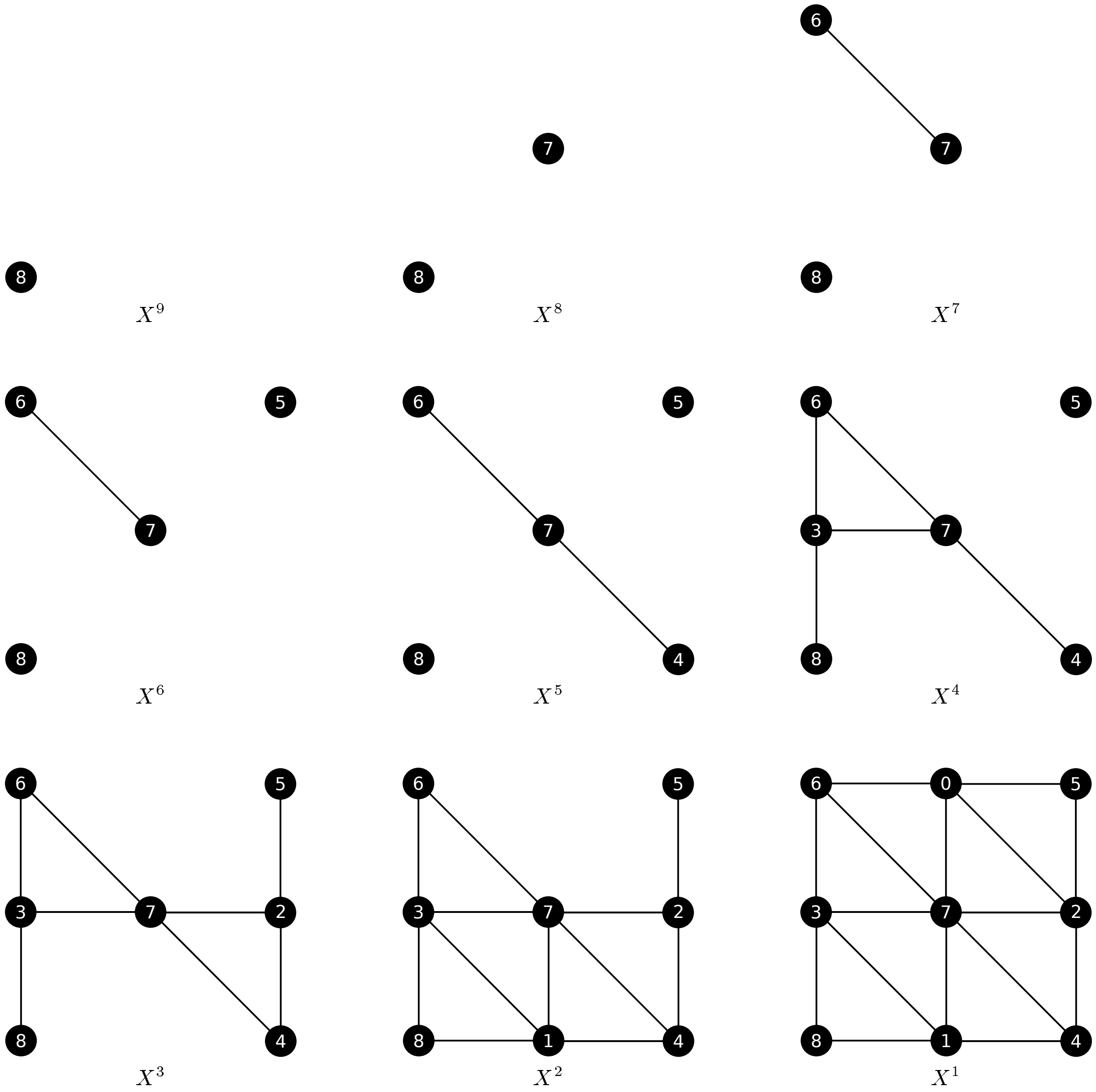
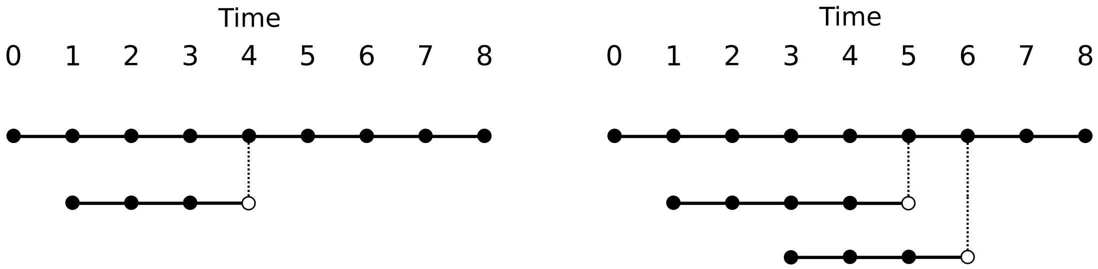


Figure 6.5: Descending Filtration of the Dataset.

We will compute the persistence pairs of the descending filtration in the same manner. Observe that one Figure 6.5 three 0th homology homology classes are born. They are born at times $0, 1$ and 3. The component 1 dies at time 5 when it is merges with the components born at time 0. The component 3 dies at time 6 when it is merged with component 0. The component 0 is the essential homology class that does not die. We can compute its extended persistence pair in a completely analogus 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 $(7, 3), (2, 5)$ and $(8, 0)$ because the vertex 7 is added at time 1, 3 is added at time 5, 2 at time 3, 5 at time 6, 8 at time 0 and 0 and time 8.

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



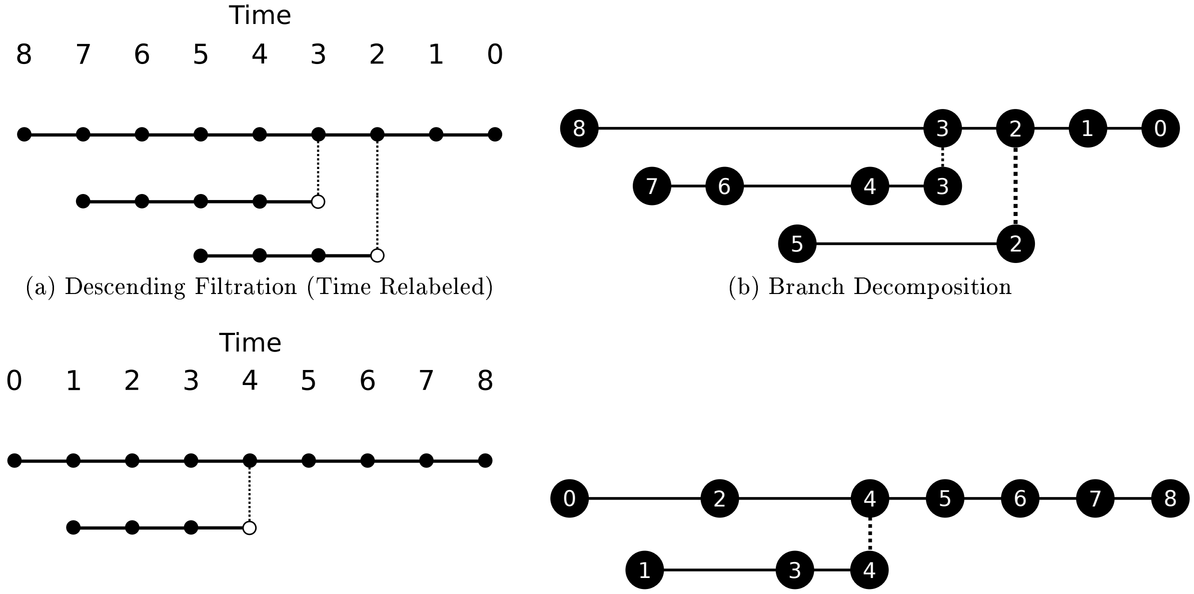
(a) Ascending Filtration

(b) Descending Filtration

Figure 6.6: Barcode Diagrams of the Persistent Homology of the ascending and descending filtration.

We are now ready to answear the primary question set forth in the begining of the section. Are the pairs of critical points produced by branch decomposition the same as the pairs produced by 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 []. The only change we have made is to relabel the time component of the descending filtration to corespond to the vertex that appears at that time.



(a) Descending Filtration (Time Relabeled)

(b) Branch Decomposition

(c) Ascending Filtration

(d) Branch Decomposition

Figure 6.7: Branch Decomposition of the split tree.

We can see that in this example the two computations are equvalent. 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 persistent homology of the descending filtration. The same holds for the split tree. This is further supported by the fact that the book [] gives an algorithm that is specific to computing the 0th persistent homology that almost identical to the algorithm for computing join/split trees.

To conclude extended persistence is not equivalent to branch decomposition of the contour tree, but there is evidence to suggest that it may be equivalent for the branch decomposition of the join and split trees. The latter remains to be proven rigorously in the general case.

Chapter 7

Empirical Study

In the final chapter we will suplement our theoretical ivestigation of the w-structures with an empirical study. The goal of this study if 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 diverse datasets. The second goal and most important goal of the empirical study is to analyse the w-structes that are present in contour trees of real life data sets using the w-diameter algorithms we've implemented. We will conclude the chapter with a discussion on the future directions this empirical study can take.

7.1 Algorithm Implementations

For the purpose of conducting the empirical study we implemented all three w-diameter algorithms we developed in Chapter []. Those are the brute force algorithm (NxBFS), the 2xBFS and DP algorithms. We implemented the brute force algorithm by running the modified version of 2xBFS from every vertex in the tree and outputting the largest w-path we found. The implementation of the 2xBFS algorithm was based entirely on the pseudocode we provided in Chapter []. Initially we made an recursive implementation of the DP algorithm based on the pseudocode we provided in Chapter []. This approach was not efficient enough because the recursion added a substantial amount of overhead in large data sets. We resorted to converting our recursive version 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 solve 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 for all 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. In order to make sure that we have solved the subproblems for all children of vertex we first root the tree using standard BFS. The root of the tree can be any vertex. Using the BFS we not only assing parrents 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 backgracking part of the DFS in the pseudocode in Chapter [].

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

All four algorithms we devloped were written in C++ and their source code can be found in their github repository [].

7.2 Data sets Overview

Before proceeding to describe 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 allows us to produce "new" data sets for testing on demand. We will now describe what algorithm we use for doing so.

Starting with a disconnected graph G with n vertices we continually generate pairs of vertices u, v in with labels in the range $\{1, 2, ..., n\}$. 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 randomly untill the 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 of 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 data set. 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. The primary use of this data set will be for the generation of a contour tree of the data and analysing the pesent w-structures. GTOPO30 however is far to large for us to handle without specialised hardware. This is why we have taken several smaller subsets of GTOPO30 provided to us by Dr. Hamish Carr.

We have chosen for our data sets to be mountaneous regious due to their more complex topographic structures. 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), icefield (240x240), pukaskwa (551x1600), gtopo30w020n40 (6000x4800). The vanc and all 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 is taken from []. Finally gtopo30w020n40 is the data set that contains all other data sets.

7.3 W-detector Algorithms

We have already provided details on the implementations of the three w-diameter algorithms we devoloped in Chapter 3. These are the NxBFS algorithm (brute force approach), 2xBFS (running modified Breadth First Search twice) and DP (dynamic programming based approach). In this section we will use their implementations 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 major issue we encountered in this test is that all three algorithm solve a problem that to our knowledge has not been been considered extensively in the past. The only way to establish ground truth on their output is to manually inspect the w-diameter of a height tree. This is neither reliable nor scalable to height trees with more than a few dosen vertices.

To overcome this issue we opted for using the output of the NxBFS algorithm as ground truth. The reasoning behind this is that it is the most straighforward to implement and that its correctness is a trivial consequence of its formulation. What this leads us to believe is that it is the most reliable of the three. Even so, a further complication arises in that the NxBFS algorithm's running time is quadratic and not linear like 2xBSF 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.

We then used this methodology to run tests on one thousand trees of sizes 50, 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10000 resulting in 10000 individual tests alltogether. The output of all tests was in line with what we predicted in Chapter 3. The algorithms DP and NxBSF had identical output and the output of the 2xBSF algorithm was no less than two of their output. These results strengthen our belief in the correctness of all three algorithms.

The second test we will perform is on the running time of the algorithms. In this test we will separate the algorithms in two groups. The first group will consist of NxBFS and the second group of 2xBFS and DP. We have separated them because we expect the running time of NxBFS to be quadratic, 2xBFS to be linear, DP to be close to linear and we want to results from each group to be comparable. The results from testing the NxBSF algorithm were completely as expected. You can see the running time on Figure []. *I willl add them if I have space otherwise not*

The tests of the 2xBFS and DP algorithm suprised. The test consisted of generating five tree on vertices in the range $\{5000, 10000, ..., 200000\}$, running both algorithm on all five trees and plotting the average running time on Figure [].

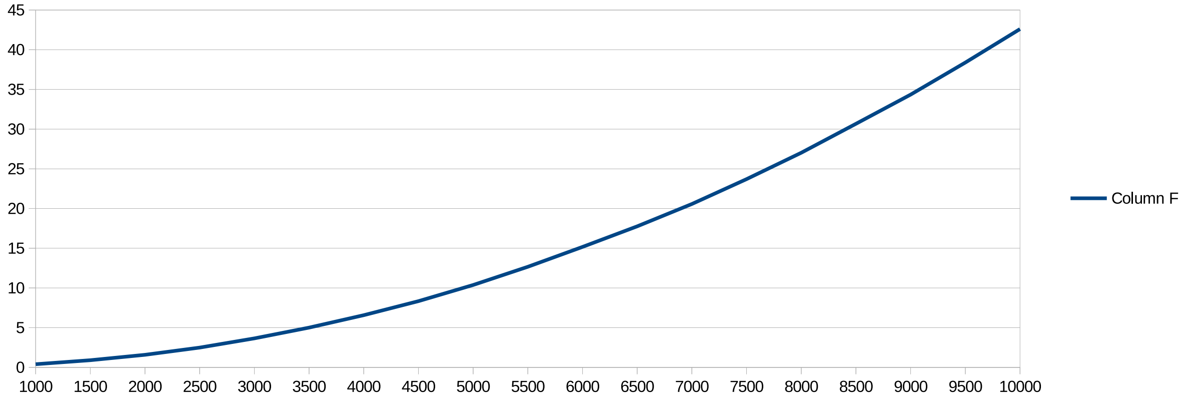


Figure 7.1: Running time of 2xBFS (blue) and DP (red) on randomly generated trees.

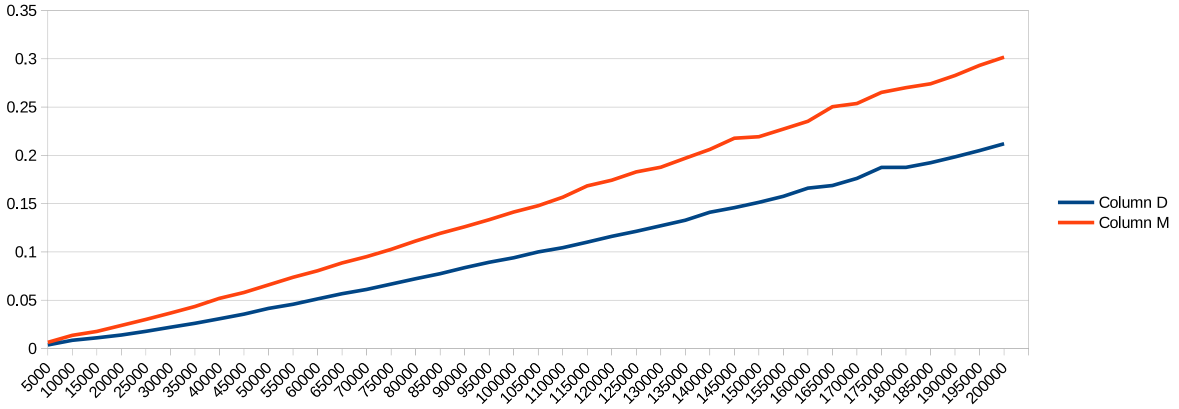


Figure 7.2: Running time of 2xBFS (blue) and DP (red) on randomly generated trees.

Both algorithms appear to fit a line almost perfectly. This is reason enough to believe that the peformance of both scale linearly with randomly generated data. This comes as no suprise as regards to the 2xBSF algorithm. 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 3 that the DP algorithm has potential for good practical performance.

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 time times faster 2xBFS is than DP in the Factor column of the tables.

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 from the previous test. As you can see the linear relationship between the two still holds. DP is within the range of 1.2 - 3 times slower than the 2xBSF 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 suprising and they requried us to double check the test multiple time. On the outlook it seems that the two algorithm had exchanged their positions. For all these unaugmented contour trees DP is 1.2 - 3 times faster than the 2xBFS algorith. We are curious to know why this is the case. The difference between the augmented and the unaugmented datasets is that in the unaugmented datasets all vertices of degree two are removed. It would appear to be the case that such vertices are processed faster by the 2xBFS algorithm and the DP algorithm has some overhead associted with them.

I'll leave this paragarph and expand it if I have extra space. Finally we would like to present a special case where the DP algorithm perform poorly. This case is for trees with vertices of high degree. Take for example a type of tree called a start Figure []. The doubly nested loop of the algorithm causes quadratic performance. To test this we generated a number of star like trees and compared the performance of NxBFS and DP.

7.4 Dataset w-diameter analysis

In this section we will examine w-structures present in real data sets. Our goal is to demonstrate that they not only pose a theoretical difficulty, but also appear in real data and hinder performace. In this study we will run our w-diameters algorithm on the mountain range data taken from GTOPO30. We will compare the w-diameters of the contour trees of the data sets with the diameters of the unaugmented and the augmented contour trees. We will demonstrate that 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 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 and shed light on whether it is the largest w-structure in a contour tree that prevents logarithmic in the merge phase. We have taken the number of iterations by running an implementation of the parallel contour tree algorithm based on [] provided to us by Dr. Hamish Carr [].

Dataset	2BFS	DP	NBFS	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

The first infrence we can make is about the pukaskwa data a 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 than we would expect that to take $log_2(881600) \equiv 19$ iterations. Instead it takes 94 iterations. This is consistent with the w-diameter of the data sets. Indeed 94 is almost twice as less as the w-diameter. If we consider that the algorithm can process two branches on oposite sides of the w-diameter at a time it makes sence that it should take twice as less iterations for the collapse if it were w-diameter that is causing it.

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 there may be some correlation between the two. In the case of pukaskwa we have already given an interpretation of this corelation. 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 in the merge phase. This is something we have not investigated as we only record the largest w-structure.

Fillay we turn our attention to the diameter of the augmented and unaugmented contour trees. The parallel contour tree algorithm currently uses those as an uppen bound on the time complexity of the merge phase []. We have already shown theoretically that the w-diameter of a height tree is necessarily smaller than its diameter. This empirical study demonstrates the practical relevance of this findind. 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. For all practical intents and purposes this is a staggering difference. If the w-diameter of that data sets were equal to the actual diameter, than we would expect the merge phase to take a far larger number of iterations and severly limit the available parallelism in it.

Chapter 8

Conclusion

In this dissertation we examined a particular tree structure that hindered parallel algorithmic performance of one of the state of the art data-parallel contour tree algorithm. These structures are long paths in height trees with a characteristic zig zag pattern. We call them the w-structures. In order to better understand them we developed three algorithms for the detection of the largest w-structure in a height tree. We prooved those algorithm are correct and showed formal bounds on their running time. Furthermore we implemented these algorithms and used them to analyse real life data sets to establish the existence of w-structures. In addition to this we explored a tools in Topological Data Analysis called Persistent Homology. This tool was primarily developed for simplifying topological features. We investigated whether Persistent Homology is equivalent to a specialised method for topological simplification of contour trees called Branch Decomposition. Through a counter example based on the w-structures we showed that they are not exactly equivalent.

8.1 Personal Reflection

The most difficult part of the project was learning the prerequisite mathematics - 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 fo 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. The most challenging part was to understand how and why exactly Extended Persistence works. It was challenging because it is a relatively new research area and there are no good unified references on the subject for students. I had to learn the mathematics behind it from reading the relevant scientific papers and then experimenting with it myself. The main difficulty was making sense of the great deal of 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. I did however had to learn how the ideas borrowed from those fields work and relate to one in other in other to pick out exactly the ones I need for my specific proofs.

My approach to learning those fields was not optimal. I overcommited a lot of my time and energy by trying to immediately learn how Extended Persistence is computed. This resulted in causing nothing but confusion and frustration. In the end 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. Afterwards I started going 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 proove. Overal I am satisfied with the results I've obtained. What was more valuable to me was learning how to teach myself new mathematics and how to approach 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 algorihm to solve it myself. 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 quite a bit of 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. Upon realising this I immediately recognised that it may be possible to modify existing tree diameter algorithms.

The first algorithm I implemented was the 2xBFS algorithm. I had not proven its correctness and I just wanted to see if it would work in practise. When I obtained satisfactory results I began to look for ways to modify the proofs of correctness. The most challenging part of constructing the proof was to disect proofs of tree diameter algorithms and see exactly which of their components I have to modify. I was puzzled when I realised that there are pathological cases where the algorithm does not return the correct answear. I initially though they could be arbitrarily bad and that would make the algorithm useless. Despite this I managed to formalise the ideas of w-path combinations and give a strict lower bound on the correctness of the algorithm.

The DP alorithm on the other hand proved to be more challenging. I could not figure out exactly how to combine paths that go through the vertex of the tree and my implementation was not returning the correct resutls. It took me alot of trial and error and manually checking the output of the algorithm on small graphs. In the initial version of the algorithm I was considering all children of children of the root and I could not determine why I was not getting the correct output. Upon formalising the algorithm and attempting the proove its correctness I realised that I had made a wrong assumption. After this I determined that I must only check the children of children that contribute to a maximum w-path.

The only part of the dissertation I that is not as well well developed is the empirical study. Initially that chapter was supposed to be the backbone of the dissertation, espcially if I had failed to produce meaningful results with the extended persistence. After commiting so much time to the w-detection algorithms and extended persistence I had already covered too much ground and had little time to produce a more detailed and insighful empirical study.

Overall the project evolved as I was working on it and my ideas for what exacly I would put in it chanded throughout the course of he summer. This approach was more risky than I would have liked, but in the end I was confident in my ability to finish it as I had planned. The project also turned out to be more theoretical than I had originally planned. I am satisfied that this is the case. This gave me the opportunity to advance my mathematical and analytical skills. It has also opened the door for more future research directions and made me more flexible.

8.2 Future Work

Finally we will present possile direction for extending this empirical study:

8.2.1 Practical

- One obvious direction is to extend the empirical study by ontaining more real life data to analyse.
- Can we obtain the w-diameter of a contour tree directly from raw data? Computing the w-diameter of the contour tree itself is useful, but it is like port mortem analysis. If we are able to obtain the w-diameter without constructing the contour tree explicitly we may be able to use the information to speed up the contruction around the w-diameter.
- Are there other ways of generating w-structures? Can we devise a general way that would work in any number of dimensions?
- We did not have a chance to investigate how different w-structures interact with one another during the merge phase. This will probably be the next best step.

8.2.2 Theoretical

There are many more unansweared question on the relations of contour tree simplification and persistent homology. For example notice that the way the 0th persistence of an ascending filtration is computed is similar to how a join tree is computer. The similarity is in that joining components correspond to connected components in sublevel sets which in turn correspond to homology classes in the 0th homology. The joining of components coresponds to merging of homology classes. Consequently the persistent pairs correspond to branches in the join tree. If we instead take a descending filtration we obtain a similarity between the split tree and the persistent homology of the descending filtration. Further work can be done in showing whether the computations are equivalent formally and in translating idea from each of the frameworks to the other.

Another question is whether contour tree simplification can be expressed through through some filtration similar to persistent homology. Consider for example a simplicial mesh M and its level sets M_i. Homology gives us to tools of identifying the connected components of the the M_i by computing the 0th homology. It is possible however to somehow relate the homology classes of different level sets. This would require a sequence of the form

$$... \to H_0(M_i) \to H_0(M_{i+1}) \to ... \to H_0(M_j) \to H_0(M_{j+1}) \to ...$$

But what would the function between the homology groups be? Can me somehow induce them through function between the sublevel sets themselvs? We certainly cannot use inclusion maps because level sets are not neseted. It is worth contemplating whether this is feasble. Another question is whether it would be practical at all. Even if we do so all we will obtain is a mathematical reformulation of a problem we already have. Will that lead to more efficient algorithms or to better understanding of the subject?

We are not able to expand more on questions like these and leave them as subject for further study.

References

[1] S. Axler. *Linear Algebra Done Right.* Springer, 3 edition, 2015.

[2] T. F. Banchoff. Critical points and curvature for embedded polyhedra.

[3] G. Carlsson. Topology and data.

[4] H. Carr. Efficient generation of contour trees in three dimensions. Master's thesis, The University of British Columbia, 2000.

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

[6] M. O. Chunyuan Li and F. Chazal. Persistence-based structural recognition.

[7] M. L. Connolly. Shape complementarity at the hemo-globin albl subunit interface.

[8] H. Edelsbrunner and J. Harer. Persistent homology - a survey. In *Surveys on Discrete and Computational Geometry*, volume 453. 01 2008.

[9] H. Edelsbrunner and J. Harer. *Computational Topology, An Introduction.* Americal Methematical Society, 1 edition, 2013.

[10] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete Comput. Geom.*, 28(4):511–533, Nov. 2002.

[11] R. Ghrist. Barcodes: The persistent topology of data.

[12] R. Ghrist. *Elementary Applied Topology.* Createspace, 1 edition, 2014.

[13] C. M. S. Hamish Carr, Gunther H. Weber and J. P. Ahrens. Parallel peak pruning for scalable smp contour tree computation.

[14] G. H. W. Hamish Carr and J. Tierny. Pathological and test cases for reeb analysis.

[15] J. S. M. v. d. P. Hamish Carr. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree.

[16] J. T. A. C. Hamish Carr, Zhao Geng and A. Knoll. Fiber surfaces: Generalizing isosurfaces to bivariate data.

[17] A. Hatcher. *Algebraic Topology.* Cambridge University Press, 1 edition, 2002.

[18] D. Kozlov. *Combinatorial Algebraic Topology*, volume 21 of *Algorithms and Computation in Mathematics.* Springer, 1 edition, 2008.

[19] C. B. V. P. M. van Kreveld, R. van Oostrum and D. Schikore. Contour trees and small seed sets for isosurface traversal.

[20] Y. Matsumoto. *An Introductino to Morse Theory*, volume 208 of *Translation of Mathematical Monograms*. American Mathematical Society, 1 edition, 2002.

[21] B. Mendelson. *Introduction to Topology*. Dover, 3 edition, 1990.

[22] J. Milnor. *Morse Theory*. Princeton University Press, 1 edition, 1963.

[23] D. Morozov and G. Weber. Distributed merge trees.

[24] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets.

[25] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. Multi-resolution computation and presentation of contour trees. 01 2004.

[26] B. H. Peer-Timo Bremer, Herbert Edelsbrunner and V. Pascucci. A topological hierarchy for functions on triangulated surfaces.

[27] M. S. B. F. S. Biasotti, D. Giorgi. Reeb graphs for shape analysis and applications.

[28] H. D. S. Maadasamy and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology.

[29] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3d in o(n log n) steps.

[30] S. K. VerovÅąek and A. Mashaghi. Extended topological persistence and contact arrangements in folded linear molecules.

[31] J. L. Yonggang Shi and A. W. Toga. Persistent reeb graph matching for fast brain search.

[32] A. J. Zomorodian. *Topology for Computing*. Cambridge University Press, 1 edition, 2009.

Appendices