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

I have come here to chew bubble gum and kick ass. And I'm all out of bubble gum.

Acknowledgements

<The page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by other to the project.>

Note that it is not acceptable to solicit assistance on 'proof reading' which is defined as the "the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the test";
see http://www.leeds.ac.uk/gat/documents/policy/Proof-reading-policy.pdf.

Contents

Chapter 1

Introduction

* —– <roadsign> Peter Construction Co. </roadsign> *

Computational Topology is an emerging field that leverages theory from topology to produce algorithms for solving various problems in structural biology, graphics, visualisation, medical imaging, X-ray crystallography and others. The mathematical field of topology is largely interested in qualitative global properties of objects. It is the natural field to study for example the "shape" of a surface or it's number of connected components.

In this dissertation we will be most interested in utilizing computational topology in the field of scientific visualisation. We shall do so through a tool that has been well established in recent years in scientific visualisation called the contour tree. The contour tree is a discrete graph data structure that is used to summarise the connectivity of * planar cross sections of a surface *. The usefulness of the contour tree is in that it can be used to identify the most topologically significant features that arise in data with little to no human interaction. Such tools become invaluable when the amount of data that can be collected far exceeds the capability of humans 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 contour tree computation. The issue current parallel contour tree algorithms have is that certain substructures we call W-Structures of the contour tree slow down computation by serialising it along these W-Structures. They are in a sense the critical path of the parallel computation and we would like to analyse them theoretically to find out why that is. We will also use the W-Structures in another theoretical setting of Persistent Homology as a counter example to a claim that has been made on the connection of contour tree computation and persistent homology computation.

The material in this dissertation is spread throughout eight chapters. The second chapter provides the reader with the necessary mathematical background to tackle the rest of the dissertation. Chapter three introduces the concept of contour trees and the state of the art parallel algorithm we have for computing them. Chapter four explores the theoretical properties of the critical substructures of contour trees we defined as W-Structures. We will develop three algorithms for detecting and analysing them. Chapter five is devoted entirely to the subfield of Algebraic Topology called Homology. In chapter six we introduce a connection between contour trees and a tool from

Computational Homology called Persistent Homology. We use this connection to tackle a claim that was falsely made in a paper that states there is a connection between the two. In the last chapter we present an empirical study on the w-structures by implementing and analysing all algorithms discussed in the dissertation. We use those algorithms to analyse real life data sets and demonstrate that these W-Structures do appear in practice and are a real problem.

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 and Algebraic Topology. 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 we shall encounter is Point Set Topology. It forms the underlying framework on top of which mathematicians build the concepts of continuous spaces and functions. As with many other mathematical disciplines topology is the study of sets that posses mathematical structure [8]. Through point set topology we can define the mathematical structure known as the topology of a set. The topology of a set aims make rigorous the notion of whether two elements of a set are "close" or "near" each other. Elements of a set which are close or near to one another are said to be a part of an open set. We can use the topology of a set to manipulate open sets by combining them to obtain new open sets. Let us now introduce the formal rules of the process.
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 topology may be $\tau = \{\emptyset, X\}$ and another may consist of all subsets of X.

Let us now introduce the topology we are going to use on $\mathbb{R}^n$. It is called the standard topology and it is based on the standard definition of distance between points in Euclidean space. Let $x = (x_1, x_2, ..., x_n)$ be a point in $\mathbb{R}^n$. Then we can define the open ball around x or 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.

But how is it possible to update this? How fast is it?

The next thing we would like to do is to define a special class of functions that preserve the properties of a topological space. 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 intuitive understanding we have of continuity from calculus - if we "slightly adjust" the output of a function in Y then there should be only a "slight change" in input in X. The "slight change" is formalised by considering all points in a single open set, as we can think of them as being "near". This is the reason why continuous functions do appear to be one "continuous whole" when plotted as in for example [].

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 the two spaces are topologically identical. *See for example [].* This is the reason why topologists are most interested in is classifying and analysing spaces up to homeomorphism. We will call a property of a space that is preserved under homeomorphisms a topological invariant.

Now let us introduce our first topological invariant - path connectedness. It captures the idea that there is a path between any points in a space. But how do we define a path rigorously?
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 $f(1) = y$.*

Paths in a topological space have a very natural geometric interpretation. * What is it? *. 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 $x, y \in X$*

This deceptively simple looking definition actually describes one of the methodologies for analysing topological spaces. Through defining auxiliary some auxiliary structure. In the case of path-connectedness we have employed a two parameter family of utility functions to "measure" a global property of the topological space - how well connected it is. The two parameter family is the collection of all paths between all pairs of points.

A very important property of path-connectedness is that the continuous image of a path-connected is path-connected. Properties like this are crucial in the task of differentiating between topological spaces. If for example one space is path-connected and another one is not there there cannot exists a homeomorphism between them. Consequently they are not homeomorphic and therefore topologically different.

Now we will present our final definition. That of a topological manifold - a mathematical generalisation of a surface.
Definition 5. *A $d-manifold$ is topological space where every point has an open neighbourhood that is homeomorphic to* $\mathbb{R}^d$.

Examples or 0-dimensional manifolds are points. Examples of one dimensional manifolds are lines, circles and graphs. Examples for 2-dimensional manifolds are the surfaces we are familiar with from geometry such as the sphere, the torus and so on. *Manifolds are the playground where topology meets geometry.* In this dissertation we will exclusively be using manifold of dimension up to two. This is because we are most interested in the visualisation aspect of computational topology and hence would like to limit ourselves to modelling spacial phenomena of up to three dimensions.

It is often hard to analyse the topology of a space by just considering it's open sets. In practice it is even computationally infeasible due to the shear number of ways we can combine open sets. This is why in the following two chapter we will employ additional tools from other fields of mathematics to aid in our analysis of the topology of a space. Two such tools are differentiable function and combinatorial approximation of spaces by decomposition into simple pieces called simplices.

2.2 Differential Topology

Differential topology is the study of differentiable functions defined on smooth manifolds. One of the leading fields of differential topology is that of Morse Theory. Morse theory is the study of the deep relation between spaces and functions defined on them. One of the main goals is to determine the shape of a shape by analysing the class of functions that can be defined on it. For example using methods from differential topology we can show that the real line and the circle are different topologically. * See Appendix for example.*

One way we can study smooth manifolds via differentiable functions is by analysing the critical points of the functions. This however is an enormous task in it's own right. This is why we will restrict ourself to a special class of differentiable functions called Morse functions.

Definition 6. *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.*

For practical considerations we will restrict our attention even further and consider Morse functions whose codomain is $\mathbb{R}$. One way we can use Morse functions defined on a manifold it to decompose it a family of nested subsets and analyse those to obtain some global topological information. We can make this concept more clear with the definition of level sets, sublevel sets and super level sets.

Definition 7. *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 intervals of the form $[-\infty, a]$ and are the preimage $f^{-1}([-\infty, a]) = \{x \in M : f(x) \in [-\infty, a]\}$. Superlevel sets are defined analogously in terms of intervals of the form $[a, \infty]$. Furthermore we shall call the path-connected components of a level set contours.

Morse functions ensures the following properties which we will make use of in the future:

- None of the critical points are degenerate.
- Changes in the topology of sub(super)level sets only happen at critical values.
- A Morse function defined on a closed surface has a finite number of critical points.

With Morse functions we are able to decompose a manifold into it's level sets. What we lack is a way to collectively analyse them and relate them to one another. This is where the concept of Reeb Graphs comes in.

2.2.1 Reeb Graph

The Reeb Graph is a tool that encapsulates the evolution of the connectivity of level sets of a continuous function. It does so by tracking how the connected connected components in the level sets appear, disappear and split or join together. 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 (when they appear, dissapear, split or join). Morse theory ensures that critical points occur at distinct values of the parameter and are isolated. This removes any ambiguities that may arise in the construction of the graph. Furthermore that fact that their number is finite on a close surface and the fact that they only happen at critical values make this computation tractable.

Definition 8. *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 $X \big/ \sim$ together with the quotient topology.*

Intuitively we can think of the Reeb Graph of the space where connected components of X are contracted to a single point. We can turn the resulting topological graph into a combinatorial structures by recording all the vertices and edges between them.

SHOW LOTS OF PICTURES

We will see in the next chapter that the Contour Tree is a special case of the Reeb Graph. Before we move on 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 it's roots in combinatorially defined topological spaces [6]. Unlike Point Set Topology and Differential Topology this allows us to obtain exact algorithm for computing the algebraic invariant we are interested in. To make matters clearer firstly we will introduce one of the most basic combinatorial topological space - 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 Five where we will see how the concept of the Euler Characteristic can be generalised to the one of the most vibrant fields of Algebraic Topology - Homology.

2.3.1 Simplicial Complexes

Simplical Complexes are the one of the first combinatorially flavoured topological spaces one encounters in Algebraic Topology. They are a set that consists of points, line segments, triangles and their higher dimensional analogues all "glued together" in a single structure. In order to understand simplicial complexes we must first understand what their basic building blocks are. The definitions here are taken from [2].

Definition 9. *Let $\{v_0, ..., v_k\}$ be points in $\mathbb{R}^d$. The convex combination of the points is the sum $\sum_{i=0}^{k} \lambda_i x_i$ where $\lambda_i \geq 0$ and $\sum_{i=0}^{n} \lambda_i = 1$.*

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

Definition 10. *Let $\{v_0, ..., v_k\}$ be points in $\mathbb{R}^{k+1}$. We will call the convex combination of those points the $k - simplex$ defined by the points.*

* Show Examples *

In the definition above the number k is also called the dimension of the simplex. Analogous to how we mentioned that we will only make use of low dimensional manifolds, here we will only be considering simplices of dimension up to three. We will call the simplices of dimension $0, 1, 2$ and 3 vertices, edges, triangles and tetrahedron respectively. See example [].

A face of a simplex is the convex hull of a non-empty subsets of it's points. Thus the points that define the simplex are it's vertices. To obtain a simplical complex all we have to do is take the union of a number of simplices in the same dimension and glue them along common faces without allowing self-intersection. More formally:

Definition 11. *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.*

Show example [].

We shall obtain the topology of a simplex through embedding it in three-dimensional Euclidean space and consider the subspace topology as a subset. Now that we have formalised the concept of a simplical complex let us see what kind of algebraic invariants we can compute from it.

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 [3]. The concept was originally defined for polyhedra as a 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. This allowed for the classification of the Platonic solids [fig].

The Euler Characteristic can be generalized to all spaces that can be decomposed into a finite number of cells. Let us first consider simplical complexes because they generalise polyhedra. The natural generalisation of the alternating sum is to continue it indefinitely with the number of 3-cells, then 4-cells, etc., as follows

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

where all k_i is the number of i dimensional simplicies and $k_i = 0$ for i bigger than some $n \in \mathbb{Z}^+$ and all k_i for $i \leq n$ are positive integers.

* Show Examples *

Lemma 1. *The Euler Characteristic is homotopy invariant.*

This result allows us to compute the Euler Characteristic of topological spaces which are not simplicial complexes. Let us take for example one of the simplest surfaces - the sphere. We will call any simplicial complex that is homeomorphic to the sphere it's triangulation. The most basic triangulation of the sphere is the tetrahedron. Therefore if we wish to compute the Euler Characteristic of the sphere all we need to do is compute the Euler Characteristic of the tetrahedron. This process is what will allows us to compute the contour tree in the next chapter. We will start with an assumption that we have a bounded volume in $\mathbb{R}^2$ and we will triangulate it to enable computation.

Chapter 3

Contour Tree Construction

The Reeb Graph of a contractable topological space is connected and acyclic []. 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 Input Data

Many scientific and medical applications require the sampling of points from a bounded area or volume in two or three dimensional Euclidean space. Such a process is for example the sampling of temperature from a bounded volume of air [] or the height of points over a geographic region []. The contour tree is shown to be a robust tool primarily used in the visualisation of such natural phenomena []. 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 fig[]). The resulting data structure we will call a simplical 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[]. 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 [] 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 [] 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 []. Those are points which are saddles and merge more than two contours. Both algorithms suffer from lack of generality and non-trivial treatment of multi-saddle points. Carr [] 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. For a more detailed overview of contour tree construction algorithms we refer the reader to [].

Finish discussion with parallel and distributed algorithms.

There are distributed algorithms. One of the data-parallel algorithms with the best performance is due to Carr. It is based on his serial algorithm. This is why we will first introduce how the serial algorithm operates. Before understanding that algorithm we need to establish some notation first. In order to discuss the inner working of the algorithm for computing contour trees we must first establish some notation and useful properties about height graphs and trees.

Also define regular points!

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 $u_i u_{i+1} \in E(G)$ 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 12. *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* $\delta^+(v) = \big|\{u \in N(v) : h(u) > h(v)\}\big|$.

The down degree of v is defined analogously as $\delta^-(v) = \big|\{u \in N(v) : h(u) < h(v)\}\big|$. 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 13. *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 example []. 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 $-f$.

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 14. *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 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 [2]. 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.

* Here is a beautiful example of this. *

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

* 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 [].

Definition 15. *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 16. *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 [] tells us that if we perform vertex contraction on v (remove it and reconnect edges) 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. This process is known as removing leaves []. For a more detailed description see [].

* See example with pretty pictures *

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 augmentation if necessary and output contour tree.

The running time of this algorithm is $O(nlogn + N\alpha(N))$.

* Add something else here *

3.6 Parallel Algorithm

The data parallel contour tree algorithm [] 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 [] 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 [] 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 fig[]). 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.

3.7 Contour Tree Simplification

Lastly in this chapter we will present Contour Tree Simplification. As we have previously shown contour trees are an established tool in scientific visualisation. A central problem in visualisation is simplifying the data and presenting only the most important parts to enable human comprehension. The contour tree of a large enough data set can quickly become an unwieldy beast in it's own right. This is why it is vital to employ techniques that simplify the contour tree by removing parts it that correspond to less "significant" topological features or sampling noise and error.

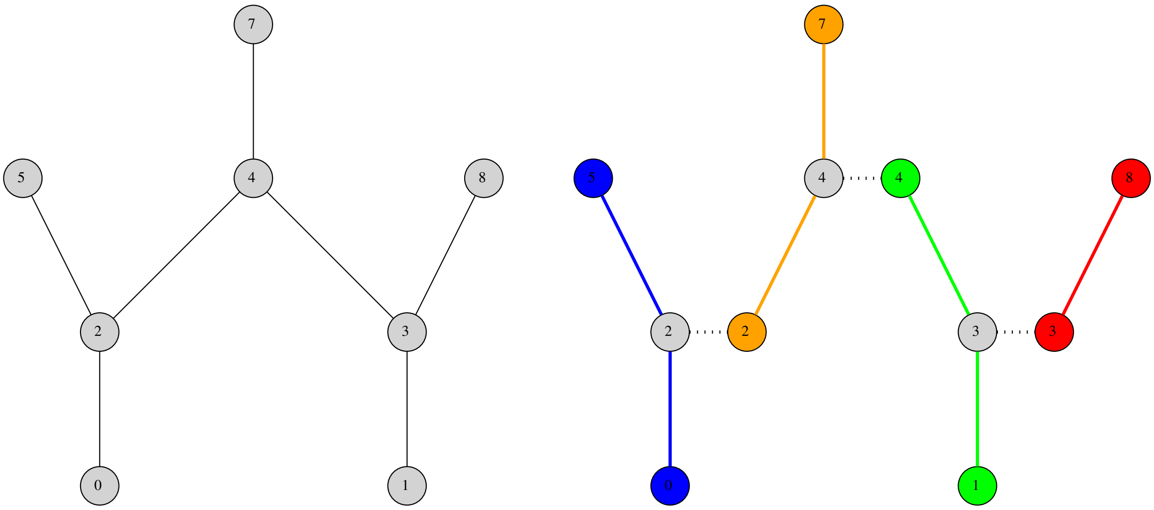
One such technique is branch decomposition [9]. Branch decomposition involves decomposing the contour tree into a set of edge-wise disjoint monotone paths which cover all edges of the tree. A trivial branch decomposition of any tree is obtained by taking every edge to be a separate branch. Furthermore 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 3.1.

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 from the tree under the following conditions:

- blq

We apply a simplification by removing a branch that does not disconnect the tree.

This produces a hierarchy of cancellations like in example [].



(a) Contour Tree.

(b) Branch Decomposition.

Figure 3.1: Branch Decomposition of a Contour tree.

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 paper [9] cites that the persistence defined in that way is similar to persistence first defined in [5]. In Chapter N of this dissertation we will demonstrate that this claim is either incorrect of misleading. Stay tuned folks.

Chapter 4

Something "W" This Way Comes!

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 we will define is their monotone path decomposition. The monotone path decompositino of a path P as a sequence of vertexwise maximal monotone subpaths of P where consecutive subpaths share exactly one vertex and have alternating direction. As shown in Figure [] P can be decomposed 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 []). 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 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.

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.

4.2 Tree Diameter Algorithms

The longest path in a graph is an *NP-hard* problem. This can be demonstrate via a reduction to the Hamiltonian path problem []. Fortunately in the special case where the graph is a tree it has a linear time solution. In this special case it is known as the tree diameter problem. In this section we will take a look at how two of the linear time tree diameter problems work.

4.2.1 Breadth First Search

The first algorithm we will discuss is based on the following theoretical result [].
Lemma 2. *Let s be any vertex in a tree. Then the most distant vertex from s is an endpoint of a tree diameter.*

To implement this algorithm we require of 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 $t \in V(T)$. As u is the farthest vertex from s it must be the endpoint of a diameter. As the diameter of T is the longest path in T therefore the second BSF must produce a path of length as much as the diameter. 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 just two consecutive times.

4.2.2 Dynamic Programming

The second approach is based on the Dynamic Programming paradigm. Dynamic Programming is a method that is used to solve optimisation problems that exhibit recursive substructures of the same type as the original problem. The key ingredients in developing a dynamic-programming algorithm are [Into to Algorithms]:

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

Trees exhibit optimal substructure through their subtrees. For our intents and purposes we shall define a subtree as a connected subgraph of a tree. We will only consider rooted trees in the context of this algorithm and we must define the accordingly. Let T be a rooted tree and let $v, u \in V(T)$ be two vertices such that v is the parent of u. We shall define the subtree rooted at u as the maximal (vertex-wise) subgraph of T that contains u but does not contain v. We will denote it as T_u. In this notation $T = T_s$ where s is the root of T. The rooted subtree at u is smaller than T as it does not contain at least one of the vertices of the T namely - v. The structure of the optimal solution is characterised through all possible rooted subtrees $\{T_u\}_{u \in V(G)}$.

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 tree either goes through two children of the root and the root or is entirely contained in one of the subtrees rooted at one the children. In order to define this formally we will make use of two additional functions. 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 define $D(u)$ as longest path contained entirely in T_u. The function we will maximize is $D(s)$ where s is the root of T. We will do so with the following formula.

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

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. We are guaranteed to reach the base cases as each subtree is strictly smaller we must inevitably reach all leaves. 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 $u \in V(T)$ []. We will elaborate more on the imeplementational details in the final section of this chapter.

4.3 Tree W-Diameter Algorithms

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 17. *(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 18. *(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.

4.3.1 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.

Lemma 3. *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:

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

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
```

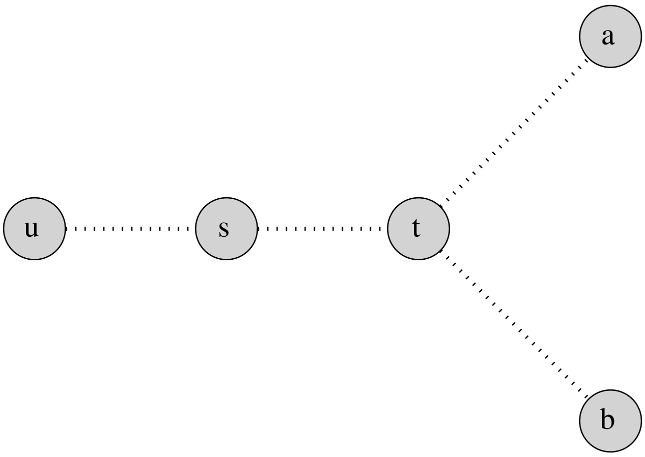


Figure 4.1: Relative position of vertices in Case 1.1

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:

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

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

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

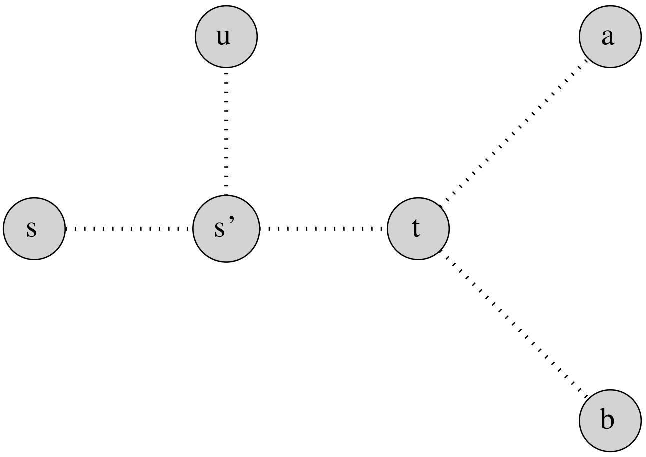


Figure 4.2: Relative position of vertices in Case 1.2

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

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. 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:

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

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

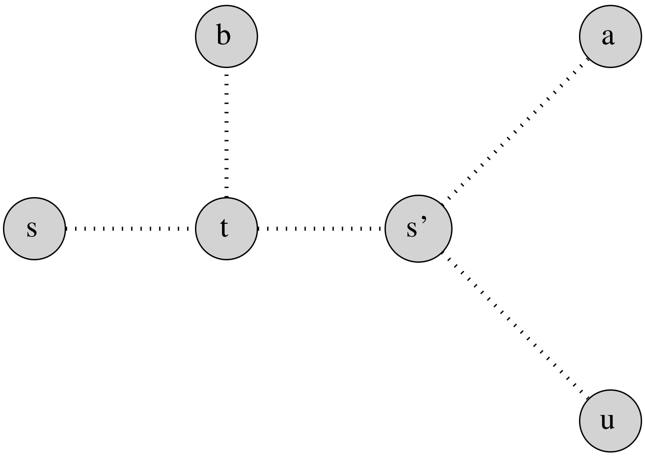


Figure 4.3: Relative position of vertices in Case 2 (t could be equal to s').

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 4. *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 5. *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 $O(|V|)$. □

4.3.2 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 4.4). Each one of the examples 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 $w(a, b) = 3$.

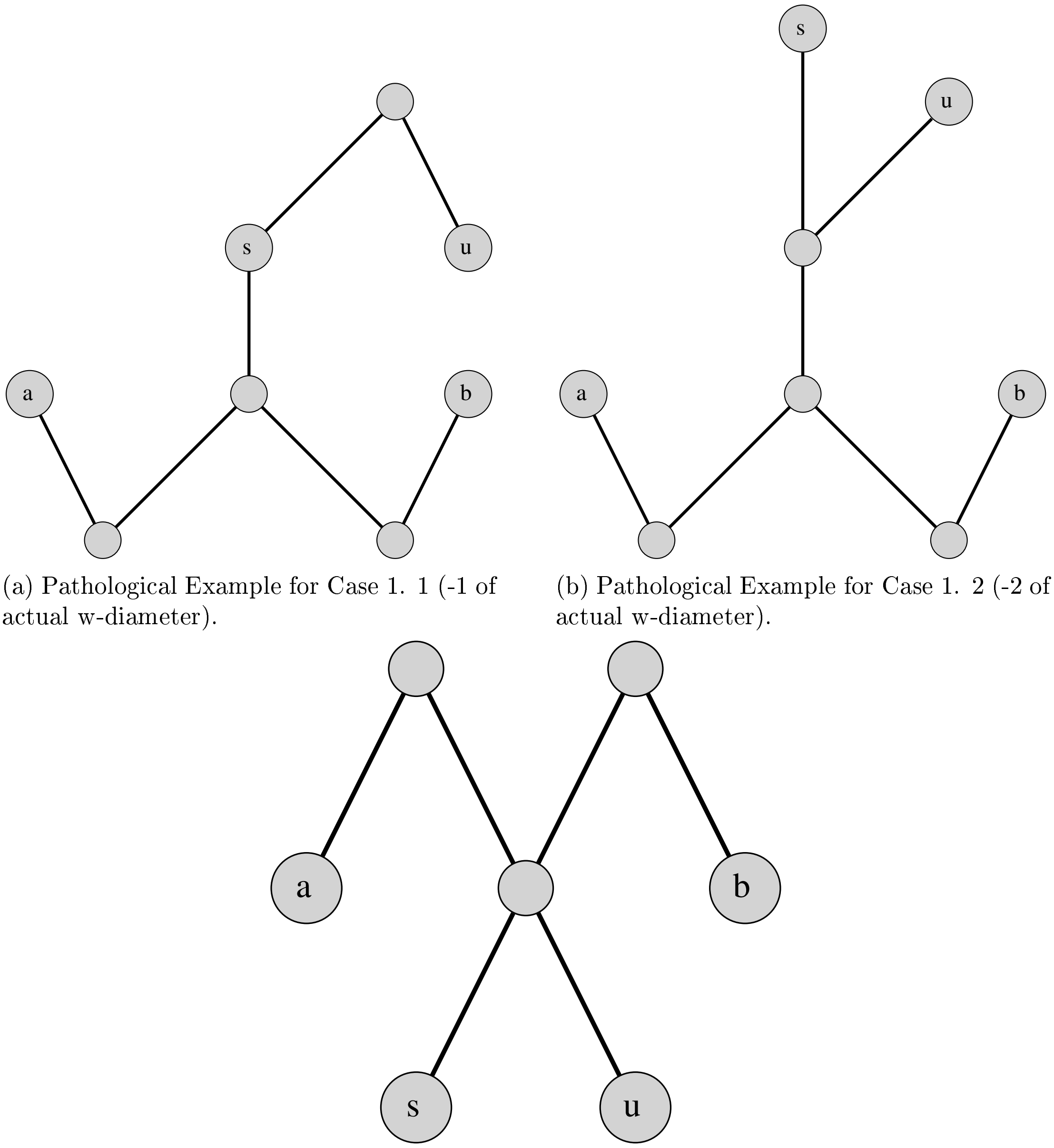
Even if we adapt the algorithm so that it finds the vertex with that is farthest in terms of both w-length and length we will still be able to make simillar pathological case examples. We just need to augment the height trees by making u further away from s that a and b but keeping the same relative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

4.3.3 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



(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 4.4: Branch Decomposition of a Contour tree.

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.

4.3.4 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:

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

Here is the pseudocode for a recursive implementation of this algorithm.

Let us now prove the correctness of the algorithm with the following Lemma.
Lemma 6. *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

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

```
 1: // Find maximum subproblem solution
 2: for all u ∈ T.Adj[s] do
 3:     O[s] = max(O[s], O[u])
 4: // Take the bigger of the two
 5: O[s] = max(O[s], maxCombine)
 6: function CALCULATE_W_DIAMETER(T)
 7:     s = <any vertex>
 8:     s.$\pi$ = s
 9:     W_DFS(T, s)
10:     return s.W
```

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 7. *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.

□

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:

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

where we use $d(u)$ for the degree of a 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:

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

When running DFS on a tree it is not possible to visit a vertex as a child of a child more than once. Suppose for the sake of contradiction that 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:

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

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.

4.4 Algorithm Implementations

For the purpose of conducting the empirical study later on we implemented all three w-diameter algorithms we developed in the this chapter. Those are the brute force algorithm, the 2xBFS and DP algorithms. For the brute force algorithm and the 2xBFS algorithms we based our implementation entirely on the pseudocode provided in the previous chapter. The source code for those can be found in the appendix. For the DP algorithm we had to implemented a bottom up approach because the recursive one we suggested in the previous chapter was not efficient enough for large data sets.

The convert the DP algorithm to a bottom up approach we first have to solve all base case subproblems and then iteratively work our way to other subproblems that depend on ones that have already been solved. In order to do this we can run a BFS from the root of the tree and label all vertices with the distance they are at from the root. We can then iterrate over all vertices in reverse order of their distance form the root. This ensures that we will solve for the leaves first and conecutively all the children of a vertex will have been solved for when we reach it. When at a vertex we can use the exact same code we provided in the backtracking part of the DFS in the pseudocode on page [].

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 [5]. 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.

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

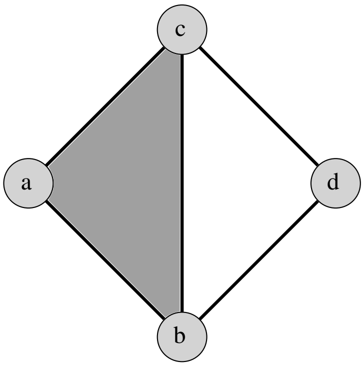


Figure 5.1: An Example Simplical Complex

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 [2]. 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:

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

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 8. *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 [4] 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 19. *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 [4] 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 [4].

- 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 [3]. 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 [2].

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 [3] 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 20. *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 [7]. 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 21. *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 22. *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 23. *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 [4]. 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).

Chapter 6

Persistent Homology and Contour Tree Simplification

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 [2]. 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). 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, apply it to contour trees and explore its relation to contour tree splification.

6.1 Persistent Homology

Persistent homology emerged in the early 2000s in the works of [5] 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 $\{X_t\}_{t \in \{0,\dots,n\}}$ 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:

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

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 [3]:

- 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 [2]. 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. Show example.

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 [2] 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 [9]. 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 [5].

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.

6.2 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 [] for more details on this.

6.2.1 Extended Persistence and Branch Decomposition

In the original paper the introduces branch decomposition of contour trees make serveral references to topological persistence and persistent homology in defining the persistence of branches.

- ". We have tested the approach using topological persistence (that is the difference in function value between a pair of critical points that are simplified) as the main metric for constructing the topological hierarchy"
- "In the next section we will discuss the construction of a hierarchical decomposition based on the persistence of critical point pairs."
- "We can now define an order on the branches that allows to extract a contour tree after any number of simplifications in linear time. First, we define the persistence of a branch to be the greater of its length and the persistence of each of it children. This definition differs from the definition of persistence given in [10] because it takes into consideration the topological obstructions. Thus a pair of critical points is never assigned a persistence value that is less than any of its obstructions."
- "Since the branches are sorted by their persistence we always draw the branches with greater persistence first (see Figures 5 and 6)."

What is not described however is how the persistence of the branches is derived formally withing the framework of persistent homology as it has been laid out in [5]. We believe that this is am important addition to the work especially since in the third quote we presented the authors have referenced one paper as "[10]" which is exactly "Topological Persistence and Simplification".

Describing formally the connection between the two in a complete and rigrous manner is beyond the scope of this dissertation. We will however pose and answer one small question that relates branch decomposition and extended persistence. Are the pairs of critical points produces by branch decomposition the same as the pairs of points produced by extended persistence of the zeroth homology? We will demonstrate that they are not will a small counter example. The counter example is based on our familliar w-structures.

Let us first begin with an example data set show in Figure []. Let us call that X. The contour tree of this data set is shown in Figure [] and a branch decomposition of this contour tree is shown in Figure []. Note that the global minimum 0 and the global maximum 8 cannot be appear in the same branch of any branch decomposition of the contour tree. There is no monotone path between them. There branch decomposition cannot produce the pair $(0, 8)$. According to [9]

This is problematic because the extended persistence of X does produce the pair $(0, 8)$ in the zeroth homology. Let us verify that. The ascending filtration of this data set is shown in Figure []. The ascending filtration consists of nine simplical complexes $\{X_1, X_2, ..., X_9\}$. According to our extended persistence computation the pair are $(1, 4)$ and $(0, 8)$. The first pair comes from ordinary persistence. We can see on Figure [] that a component is born in time 1 and dies in time 4. The second pair is of the global minimum and global maximum. It comes from extended persistence. To verify this observe that $H_0(X_9) = \mathbb{Z}_2$ because $X_9 = X$ has one connected component. The next homology group in the sequence is $H_0(X, X^9)$. From the Excision Theorem we have that $H_0(X, X^9) = \widetilde{H}_0(X/X^9)$. However $X/X^9 = X/\{9\} = X$ because quotienting by a single points leaves the complex unchanged. Therefore $H_0(X, X^9) = \widetilde{H}_0(X)$. We have already explained that $H_0(X) = \mathbb{Z}_2$, so $\widetilde{H}_0(X) = 0$ and consequently $H_0(X, X^9) = 0$. This means that the induced map $i_* : H_0(X_9) \to H_0(X, X^9)$ is the zero map. The conclusion we draw is that the homology class that is born at time 0 at the global minimum dies at time 8. Extended persistence produces the pair pair $(0, 8)$.

Note that this is the case for the descending filtration or for that matter for any filtration of any path connected simplicial complex. Let us demonstrate this. Let M be path connected simplicial complex. Then $H_0(M) = \mathbb{Z}_2$. Let $H_0(M, M')$ be any of the groups in the relative sequence where M' is a subcomplex of M. By Excision we have that $H_0(M, M') = \widetilde{H}_0(M/M')$. From topology we know that [] a quotient space of a path connected topological space is path connected. Therefore $H_0(M/M') = \mathbb{Z}_2$ and $\widetilde{H}_0(M/M') = 0$ accordingly. We have thus shown that in the extended persistence of a path connected simplicial complex the global minimum pairs with the global maximum.

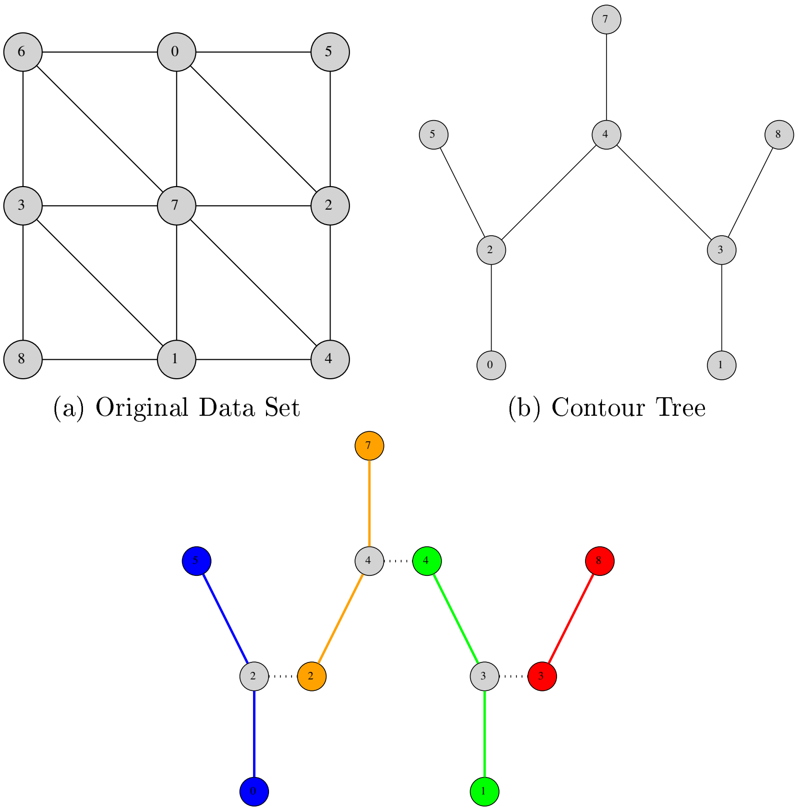
Here is the summary of our findings.

- There is no monotone path between the global minimum and global maximum in the simplicial mesh.
- There is no monotone path between the global minimum and global maximum in the contour tree.
- No branch decomposition of the contour tree can pair the global minimum and the

global maximum.

- Extended persistence pairs the global minimum and the global maximum.

This counter examples demonstrates that branch decomposition does not yield pairs of critical points that are equivalent to extended persistence. In the next chapter we will discuss what other future question we can pose.



(a) Original Data Set

(b) Contour Tree

(c) Branch Decomposition.

Figure 6.1: Branch Decomposition of the Contour tree.

6.2.2 Thoughts on future directions

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.

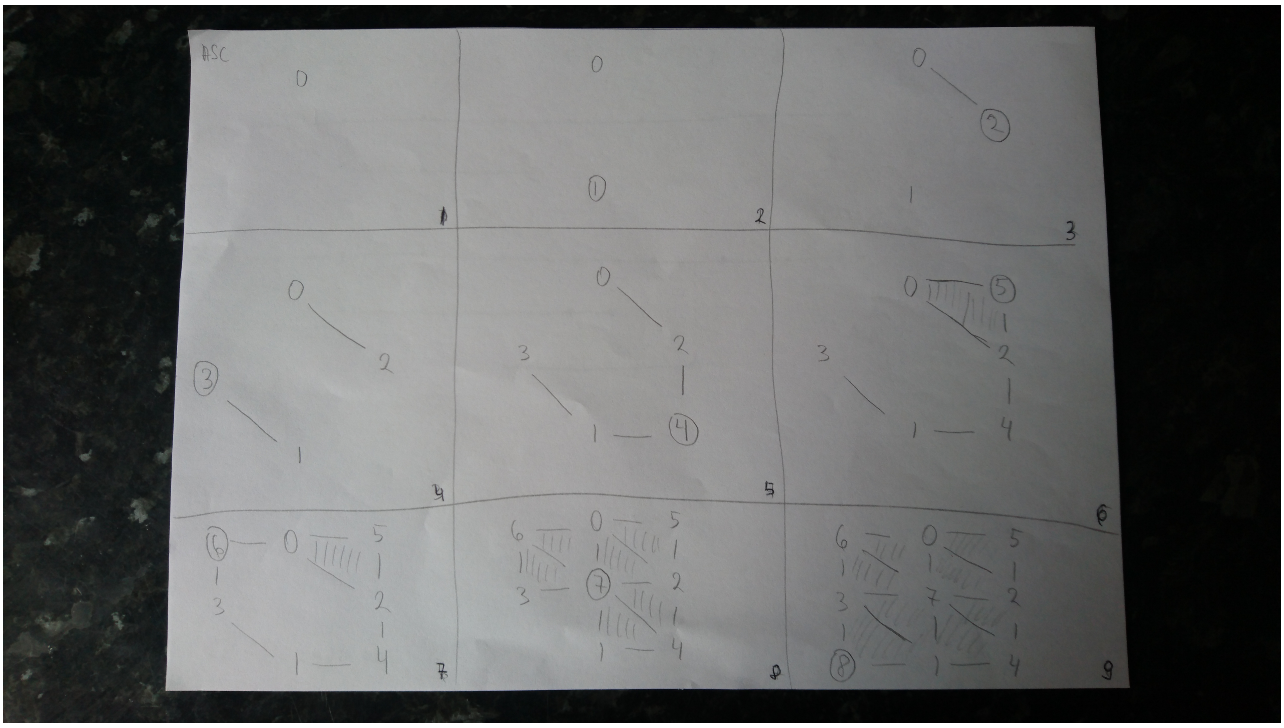


Figure 6.2: Ascending filtration of the Simplical Complex

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

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

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.

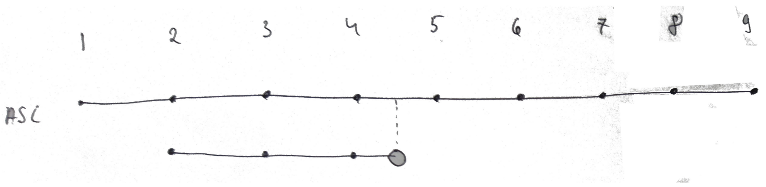


Figure 6.3: Extended Persistence of the Simplical Complex

Chapter 7

Empirical Study

In our final chapter we will suplement our ivestigation of the w-structures with an Empirical Study. This goal of this study if two-fold. Firstly to verify our theoretical claims on the correctness and running of the w-diameter algorithm we developed. This will be done by implementing and testing them. The second goal and most important goal of the empirical study is to analyse w-structes in contour trees of real life data sets using the algorithms we've implemented. We will then conclude the chapter with a discussion on the future directions this study can take.

7.1 Datasets Overview

Talk about the data sets we used. Mountain ranges taken from Canada.

7.2 W-detector Algorithms

We outlined how the algorithms are imeplemented in chapter 3. Here we will talk about testing correctness and running time.

To run tests we had to generate random height trees.

Tests correctness by using the 2xBSF algorihtm for groups truth. Easiest and most reliable. Generate a bunch of trees and check results. We test this many trees and DP had the same results, 2xBSF was always within two of the actual w-diameter. This is completely in line with what we expected.

Testing running by running on random trees in the range 5K-200K. Five different trees per mark averaged out.

Bot algorithm appear to be linear on this scale and these trees. These begins to confirm the suspisions we had the DP is not quadratic in practice.

Here is the performance on the data sets we have.

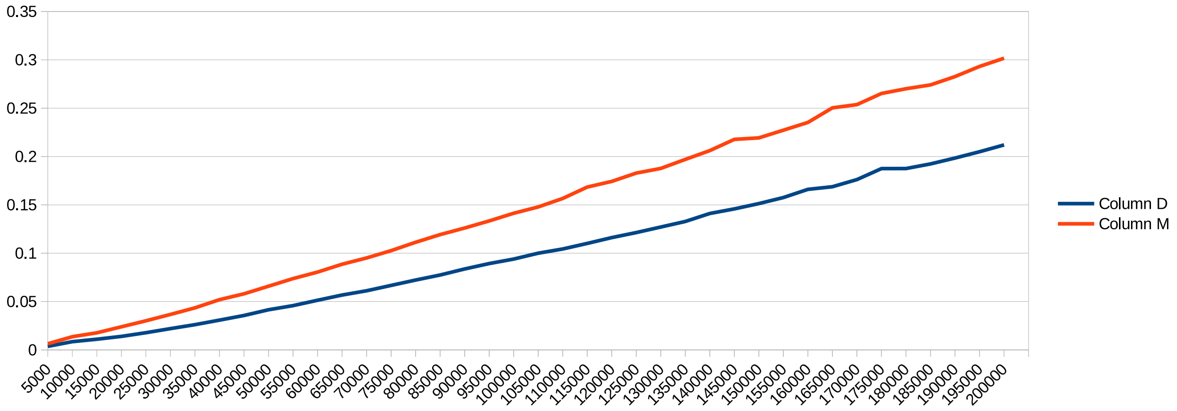


Figure 7.1: Running time of 2xBFS (blue) and DP (red) on randomly generated trees.

Dataset	Vertices	2xBFS	DP	Factor
vanc	378	0.000254	0.000477	1.88
vancouverNE	4851	0.003125	0.006393	2.05
vancouverNW	4900	0.003165	0.006536	2.07
vancouverSE	4950	0.002927	0.008382	2.86
vancouverSW	5000	0.003119	0.006285	2.02
vancouverSWNE	1250	0.000784	0.001408	1.80
vancouverSWNW	1250	0.000862	0.001490	1.73
vancouverSWSE	1275	0.000815	0.001651	2.03
vancouverSWSW	1225	0.000814	0.001412	1.73
icefield	57600	0.040002	0.070194	1.75
pukaskwa	881600	0.551351	0.973880	1.76
gtopo30w020n40	28800000	-1	-1	-1

As we can see the results are again consistent with what we had with the random data sets.

7.3 Dataset w-diameter analysis

Here is the epic table for the analysis of the datasets. Fix it it's wrong!

Dataset	Vectices	2BFS	DP	NBFS	Diameter	Iterations
vanc	378	2	2	2	311	2
vancouverNE	4851	4	5	5	1338	5
vancouverNW	4900	5	5	5	1456	5
vancouverSE	4950	6	6	6	1306	5
vancouverSW	5000	4	4	4	1977	4
vancouverSWNE	1250	5	5	5	423	4
vancouverSWNW	1250	3	3	3	712	3
vancouverSWSE	1275	3	3	3	759	3
vancouverSWSW	1225	2	2	2	845	3
icefield	57600	7	7	7	12280	6
pukaskwa	881600	180	182	N/A	374866	94
gtopo30w020n40	28800000	8	10	N/A	15766966	8

We can observe the following things:

- Things and stuff1
- The w-diameter of a tree is a much better upper bound on the algorithm.
- The w-diameter can severely prevent logarithmic collapse.
- The w-diameter becomes prevalent in random samples of randomly generated tree. Therefore the law of large number will affect it.

7.4 Generating W-Structures Manually

We can generate them via ridges and valleys.

7.5 Future Work

Can be obtain the w-diameter from raw data?

Can we generate them in other ways?

Chapter 8

Conclusion

References

[1] S. Axler. Linear algebra done right.

[2] H. Edelsbrunner and J. Harer. Computational topology, an introduction.

[3] R. Ghrist. Elementary applied topology.

[4] A. Hatcher. Algebraic topology.

[5] D. L. Herbert Edelsbrunner and K. Cole-McLaughlin. Topological persistence and simplification.

[6] D. Kozlov. Combinatorial algebraic topology.

[7] D. Kozlov. Combinatorial algebraic topology.

[8] B. Mendelson. Introduction to topology.

[9] K. C.-M. V. Pascucci and G. Scorzelli. Multi-resolution computation and presentation of contour trees.

Appendices

Appendix A

Additional Proofs

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

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

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

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

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

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

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

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

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

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

Combining this with the previous equation we obtain that:

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

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

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

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

Which is exactly what we set out to proove.

□

Lemma 10. *There are at least k vertices for every vertex of degree k in a tree.*

Proof. Let T be a tree and $u \in V(T)$ be a vertex in it. As any tree can be rooted, let us root T at u and call the new directed tree T_u. Let $U = \{u_1, u_2, ..., u_k\}$ be the neighbours of u. For each $u_i \in U$ if u_i is not a leaf let u_i be one of it's children. Repeat this process until every u_i is a leaf. This is possible because T is finite. All of the u_i are distinct, for otherwise there would be a cycle in T.

□

Appendix B

Vector Spaces, Quiver Diagrams and Barcode Diagrams

This chapter will probably be redistributed in the homology chapter. I'll probably remove it.

Should I define a vector space, bases, etc.?

Should I define a vector space, bases, etc.?

Suppose we have a number of vector spaces $(V_1, V_2, ..., V_n)$

Suppose we have a number of vector spaces $(V_1, V_2, ..., V_n)$ together with linear maps $(f_1, f_2, ..., f_{n-1})$ that that map between consecutive vector spaces like follows : $f_i : V_i \to V_{i+1}, \forall i = 1, 2, ..., n-1$.

A quiver representation is a directed multigraph where the vertices are sets and directed edges are function between sets. In our case the vertices will be vector spaces and the edges linear maps. The quiver diagram of the configuration we just described looks as follows:

$$V_1 \xrightarrow{f_1} V_2 \xrightarrow{f_2} ... \xrightarrow{f_{n-1}} V_n$$

"This sounds weird, fix it." Not that we can always extend any sequence of vector spaces with the null vector space and the null maps as follows:

$$0 \longrightarrow ... \longrightarrow 0 \longrightarrow V_1 \xrightarrow{f_1} V_2 \xrightarrow{f_2} ... \xrightarrow{f_{n-1}} V_n \longrightarrow 0 \longrightarrow ... \longrightarrow 0$$

A barcode diagram is a digram that shows which shows how the basis elements of the vector spaces evolve as they get mapped through the linear functions once we commit to particular basis elements for each vector space.

Show a barcode diagram.

A Chain Complex is a quiver representation where the image of each maps is a subset of the kernel of the next one.

$$... \longrightarrow V_1 \xrightarrow{d_1} V_2 \xrightarrow{d_2} ... \xrightarrow{d_{n-1}} V_n \longrightarrow ...$$

This example is a chain complex when $im(d_k) \subseteq ker(d_{k+1})$. As the image is a subset of the kernel the we can equivalently write this as the composition $d_{k+1}d_k = 0$. In practical terms once we commit to baseis multiplying consecutive matricies will equal the zero matrix. An important property of the barcode diagram of chain complexes is that no line can be longer than two units!

Example 1. *A Simple Chain Complex*

Let us now for simplicity and demonstrational purposes assume that each V_i is isomorphic to $\mathbb{R}^n$ for some $n \in \mathbb{Z}$.

An exact sequence is a chain complex where $im(d_k) = ker(d_{k+1})$. Exact sequences are useful because of the nice properties like ...

The homology of a chain complex is defined as a quantifier of how far a chain complex is from being an exact sequence. It is defined as: $H_k = ker(d_{k+1})/im(d_k)$

Let V be a vector space and W a subspace of V. A coset of W is the set $v + W = \{v + w : w \in W\}$.

A quotient in a vector space is defined in the following way:

$$V/W = \{v + W : v \in V\} = \{\{v + w : w \in W\} : v \in V\}$$

Show a picture of the cosets.

Luckily in $\mathbb{R}^n$ we have the following theorem: $\mathbb{R}^n/\mathbb{R}^m \simeq \mathbb{R}^{n-m}$ where we have slightly abused notation as $\mathbb{R}^m$ can not be a subspace of $\mathbb{R}^n$, but we consider it isomorhpic to one for $m \leq n$.

$$\begin{tikzcd}\mathbb{R}^3 \arrow[r] & \mathbb{R}^2 \arrow[r] & \mathbb{R}^4\end{tikzcd}$$

Appendix C

External Material

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Appendix D

Ethical Issues Addressed

Appendix E

Topologies on $\mathbb{R}$ and $\mathbb{R}^n$

Example 2. *The standard topologly on* $\mathbb{R}$*.*

The standard topology on $\mathbb{R}$ is build from subsets of $\mathbb{R}$ called open balls. The open ball centered at $x \in \mathbb{R}$ with radius $\epsilon \in \mathbb{R}^+$ is a subset $B_\epsilon(x)$ of $\mathbb{R}$ defined as:

$$B_\epsilon(x) = \{y \in \mathbb{R} : |x - y| < \epsilon\}.$$

These are all the points whose distance from x is less than ϵ. The collection of all open balls as x ranges over $\mathbb{R}$ and ϵ ranges over $\mathbb{R}^+$ makes up the building blocks of the topology. The open sets in the topology are all the open balls together with their arbitrary unions and finite intersections.

Example 3. *The standard topologly on* $\mathbb{R}^n$*.*

We can slightly adjust the previous definition to obtain a topology on $\mathbb{R}^n$. We just have to consider $\vec{x}$ and $\vec{y}$ to be vectors in $\mathbb{R}^n$ and evaluate the distance between them using the standard Eucledian metric. That is if $\vec{x} = (x_1, ..., x_n)$ and $\vec{y} = (y_1, ..., y_n)$ then:

$$B_\epsilon(\vec{x}) = \{\vec{y} \in \mathbb{R}^n : \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2 < \epsilon}\}$$

is a subset of $\mathbb{R}^n$ with all points of distance less than ϵ from $\vec{x}$. Like previously the topology on $\mathbb{R}^n$ is obtained through arbitrary unions and finite intersections of the set of open balls.

Appendix F

Circle and Real Line

Consider for example the real line $\mathbb{R}$ and the circle S^1. There are differentiable functions from $\mathbb{R}$ to $\mathbb{R}$ such as $y = x$ which do not take a minimum or a maximum value. They can be arbitrary large or small on the manifold $\mathbb{R}$. It is not possible to define such a differentiable function from S^1 to $\mathbb{R}$. This is due to the maximum value theorem. More formally a differentiable function is continuous and S^1 is compact. By [] the continuous image of a compact space is compact and by [] the compact spaces of $\mathbb{R}$ are closed and bounded. Closed and bounded means unions of intervals of the form $[a, b]$ where $|a|, |b| < \epsilon$ for some $\epsilon \in \mathbb{R}$. We can pick the lower bound of the interval with the lowest lower bound and the upper bound of the interval with the highest upper bound for the minimum and maximum values. Therefore any differentiable function defined on S^1 will take a minimum and a maximum value.