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

Talk about why parallel algorithms are important

Talk about the beauty of Computational Topology and why it's useful and where it's used.

The mathematics covered in this dissertation are far too broad to be presented in all their magnificence. This is why rather than attempting to introduce the theory in the classical textbook fashion of definition-theorem-proof I have opted out for focusing more on developing intuition behind the big ideas at play. I do so because I will later rely on the reader's intuition in presenting examples and the further technical developments of the subject of computational topology in the recent years.

1.1 Set Theoretic Notation

Definition 1. *Let X, Y be two sets and f be a function between them. Let $A \subseteq Y$ the preimage of A under f is defined as the points in X which are mapped onto A. It is denoted as $f^{-1}(A) = \{x \in X : f(x) \in A\}$*

Note that taking the preimage of a set does not require f to be invertable.

1.2 Point Set Topology

Topology is the mathematical field that studies continuous change between topological spaces. Any set X can be a topological space as long as we defined a collection of subsets of X called open sets. The open sets represent elements of X which are "near" or "close" to one another. If we have two topological spaces X an Y and wish to study how one can be continuously mapped to the other we instead focus on how the open sets are mapped. The open sets provide certain structure on the sets and we would like to study the functions that preserve that structure and focus on the properties of spaces that are invariant under such functions. Those functions are what we call the continuous functions.

Let us now be more formal now and define these notions precisely.

Definition 2. *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 $\bigcup_{\lambda \in \Lambda} U_\lambda \in \tau$.

Definition 3. *Let (X, τ) be a topological space. Any subset of $A \subseteq X$ which is in τ is said ot be open.*

Definition 4. *Let (X, τ) be a topological space. Let $x \in X$ be any element of X. We will call x a point in X and we will call any open set A containing x an open neighbourhood of x.*

The pair (X, τ) is called a topological space. In practice one builds a topology by first figuring out how they would like their open sets to look like and then takes all unions and finite intersections to obtain the full topology.

Example 1. *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 2. *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.

One may notice that the topology we put on a set is by no means unique. If we wish we

may even use the topology made up of *all* subsets of a given set. That topology is not preferred because introduces very little structure to our topological space. As we will shortly see it makes the question of continuity rather moot. Topologist prefer topologies that introduce structure on a space that is both intuitive and reflective of the properties they wish to study. The standard metric is useful because Eucledian distance is the natural quantifier of how "near" things are in almost all applied mathematical models. While the information about the actual distance is lost in the generality of the topology, the structure it imposes allows us to talk about important global properties of spaces such as path-connectednes and compactness.

Having a topology on $\mathbb{R}^n$ is all well and good but in this dissertation we shall work with surfaces and triangulations of surfaces in $\mathbb{R}^2$ and $\mathbb{R}^3$. If we had to define a topology on them in a similar fashion we would not get far. Luckily subsets of topological spaces can naturally inherit the topology of their superset.

Definition 5. *Let (X, τ_X) be a topological space and A be a subset of X. The subspace topology of A is defined as:*

$$\tau_A = \{U \cap A : U \in \tau_X\}.$$

To obtain all open sets of A take the open sets in X and remove from them all points which are not in A. Going further we will consider any surface embedded in $\mathbb{R}^2$ and $\mathbb{R}^3$ to have the subspace topology of the standard topology unless otherwise stated.

We are finally ready to present the definition of a continuous function.

Definition 6. *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 - 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 are colloquially described as manipulating an object in space without glueing together parts of it or making holes in it. Those disrupt the open sets.

So far we have obtained the means of endowing any subset of $\mathbb{R}^n$ with a topology and we have outlined a general recipe for creating continuous function between them - have open sets be the preimage of open sets. We will now introduce our first topological invariant. But first we shall show how we can "move" around in a topological space.

Definition 7. *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 $f(1) = y$.*

This is analogous to the definition of a close curve from differential geometry. The main difference being that we have no notion of differentiability or smoothness yet. As an example think of the space X as a surface in $\mathbb{R}^3$ and two distinct points x and y on it. A path between x and y is a curve that starts at x, moves around the surface and ends at y.
Definition 8. *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 holds the overarching methodology for reasoning about algebraic invariants of topological spaces. In this case we have employed a two parameter family of utility functions to "measure" a global property of the topological space. The two parameter family is the collection of all paths between all pairs of points.
Proposition 1. *The continuous image of a path-connected space is path-connected.*

Notice that in this definition we have implied surjectivity. Indeed if X, Y are topological spaces such that X is path-connected and Y is not and $f : X \to Y$ is a continuous function then all we can say is that $im(f)$ is path-connected.

Lastly we must explore how topological spaces are viewed by topologist. After all if two spaces share all of their topological properties should they be considered different? See coffee mug and dognut example []. We shall make this precise with the notions homeomorphism and homotopy equivalence.
Definition 9. *Two topological spaces X and Y are said to be homeomorphic when there exists a bijetion $f : X \to Y$, such that f and f^{-1} are continuous. Furthermore the function f is called a homeomorphism between X and Y.*

Homeomorphism is the appropriate equivalence relation for topological spaces. For example if we have two homeomorphic topological spaces X and Y and we know one of them has a particular topological property, then the other one will have it as well. Conversely if we know that a space is path connected and another one isn't, then they are not homeomorphic.

In the realm of algebraic topology there is another equivalence relation that is more flexible than this and still preserve the algebraic invariants of spaces. Before presenting it we must first introduce homotopy of functions.
Definition 10. *Let X and Y be two topological spaces. Let $f, g : X \to Y$ be two continuous functions. We say that f and g are homotopic when there exists a third continuous function $H : X \times [0, 1] \to Y$ such that:*

- $H(x, 0) = f(x), \forall x \in X$
- $H(x, 1) = g(x), \forall x \in X$

We can think of homotopy as a one parameter family of functions $\{f_t\}_{t\in[0,1]}$ such that

$f_0 = f$ and $f_1 = g$. Homotopy defines an equivalence relation on all continuous functions between X and Y. This notion can be extended to topological spaces as follows:
Definition 11. *Two topological spaces X and Y are said to be homotopy equivalent if there exist two continuous functions $f : X \to Y$ and $g : Y \to X$ such that:*

- $f \circ g$ is homotopic to i_X
- $g \circ f$ is homotopic to i_Y,

where i_X and i_Y are the identity functions on X and Y respectively.

Intuitively we can think of homotopy as a continuous deformation of the image of one of the functions to the image of the other. One can readily observe that every homeomorphism is a homotopy equivalence. Let $g = f^{-1}$, then $f \circ g = i_X$ and $g \circ f = i_Y$ as f is a bijection and every function is homotopic to itself. However the converse is not true and a homotopy equivalence is not a homeomorphism.

In a way very similar to abstract algebra continuous function play the role of homomorphisms and homeomorphisms play the role of isomorphisms. Also note that homotopy equivalence does not necessarily preserve all topological properties of a topological space. They do however preserse those which we care about such as path-connected and algebraic structures defined on the topological spaces. We will prefer to use them instead of homeomorphisms.
Definition 12. *Topological Invairant*

I will add more things to this chapter later if I need to

1.3 Building Blocks

The most basic building blocks in Computational Algebraic Topology are Abstract Simplical Complexes (ASC) and Simplicial Complexes (SC). Abstract Simplical Complexes are a combinatorial structure aimed at generalising discrete graphs. Simplicial Complexes are their geometric realisation in standard Eucledian Space. For example a planar graph $G = (V, E)$ as an ASC as a collection of vertices and edges. An embedding in the plane with points representing vertices and line segments (or curves) between them as edges is the SC.

Show Example

Let us introduce these concepts more formally starting with ASC and then demonstrating how they can be realised in Euledian Space via SC.
Definition 13. *Given a set V, an Abstract Siplicial Complex called Δ on V is a collection of subsets of V such that if $\sigma \in \Delta$ and $\varphi \subseteq \sigma$ then $\varphi \in \Delta$.*

In the spirit of keeping our definitions computationally tractable we will only consider ACS of finite sets. Elements of Δ are called simplices. The dimension of a simplex is it's cardinality minus one. We will denote by $\Delta_n \subseteq \Delta$ all simplices in Δ of dimension n. To connect this to geometry simplices of dimension zero are called vertices and simplices of dimension one are called edges. To generalise further two dimensional simplices are called triangles and three dimensional simplices are called tetrahedra. Simplices of higher dimension lose their geometric flavour, so will avoid naming them altogether.
Example 3. *Show a simple example of an abstract simplical complex.*
Definition 14. *Let σ be an n-simplex in Δ. A face of σ is any simplex φ in Δ such that $\varphi \subseteq \sigma$.*

We will call the faces of a simplex that have k dimensions less it's codimension k faces. The codimension one of a tetrahedra are the triangles. The codimension two faces are the edges and so on.

An ASC is closed under taking subsets, so the faces of a simplex are in fact all subsets of the simplex. This is also one of the reasons why high dimensional ASC are avoided in practise. For example an n-dimensional simplex has 2^n faces in the complex.
Definition 15. *Let V be a finite set and Δ an ASC of V. Let $\sigma \in \Delta$. The star of σ is the set of all simplices of which σ is a face. In set theoretic notation:*

$$St(\sigma) = \{\varphi \in \Delta : \sigma \subset \varphi\}.$$

* REDO THIS OR LEAVE IT OUT* The interplay between continuous mathematics and combinatorics is indeed interesting. For example in this context the star of a vertex plays the role of an open neighbourhood. We can define a topology on an ASC called the Alexandroff topology.
Definition 16. *The collection of all start in an ASC Δ is basis for a topology. The topology is finitely generated by $\{St(\sigma)\}_{\sigma\in\Delta}$.*

While ASC are a purely combinatorial construct, like graphs they do have a geometric realisation.
Definition 17. *The standard geometric n-simplex is the convex hull of the set of endpoints of the standard basis $[1,0,...,0],[0,1,...,0],...,[0,0,...,1]$ in $\mathbb{R}^{n+1}$ defined as:*

$$\Delta^n = \{(t_0, t_1, ..., t_n) \in \mathbb{R}^{n+1} : \sum_{i=0}^{n} t_i = 1 \text{ and } t_i \geq 0, \forall i = 0, 1, ..., n\}$$

We will define the face of an n-simplex analogously as:
Definition 18. *A face of the standard geometric n-simplex is the convex hull of a subset of endpoints of the standard basis $[1,0,...,0],[0,1,...,0],...,[0,0,...,1]$ in $\mathbb{R}^{n+1}$ defined as:*

Example 4. *Show an example of the basic simplices.*

A Simplical Complex K is the finite collection of homeomorphic images of standard simplicies such that:

- If the simplex σ is in K then all of it's faces must also be in K.
- The intersection of the images of two simplicies is either empty or a common face of both.

You can imagine using the standard simplicies as building blocks for a Simplical Complex. We take them, embed them in a high enough dimensional Eucledian space and glue them together along their common faces. Self intersections are of course not allowed. Furthermore every ASC has a geometric realisation, but not necessarily in their original dimension [2].
Theorem 1. *Every abstract simplicial complex of dimension n has a geometric realization in* $\mathbb{R}^{2d+1}$.

1.4 Vector Spaces, Quiver Diagrams and Barcode Diagrams

THIS CHAPTER IS VERY MUCH UNDER CONSTRUCTION

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 5. *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 $m \leq n$.

$$\mathbb{R}^3 \longrightarrow \mathbb{R}^2 \longrightarrow \mathbb{R}^4$$

1.5 Algebraic Topology

Algebraic Topology concerns itself with topological invariants of algebraic nature. These invariants are obtained through the extraction of algebraic structures like groups, rings, vector fields or modules from a topological space. The fact that they are topological invariants is demonstrated by the fact that a homeomorphic image of that space produces an isomorphic algebraic structure of the same kind. In practical terms these algebraic structures measure properties of a topological space such as connected components, number of holes (1-cyles), number of voids (2-cycles) or more generally number of n-cycles. This section will focus on developing the mathematical apparatus by which homology is built.

1.5.1 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 suitably nice spaces through a generalisation of counting [4]. 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 CW-complexes because they generalise polyhedra. The natural generalisation of the alternating sum is to continue it indefinitely by 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 $k_i = 0$ for $i > n$ and all k_i for $i \leq n$ are positive integers. This sum of course works perfectly fine for simplical complexes as well.

Even more generally given a topological space X that can be decomposed into the disjoint union of a finite number of open cells $X = \coprod_\alpha \sigma_\alpha$ where each k-cell σ_α is homeomorphic to $\mathbb{R}^k$ we can apply the same formula as above [4].

$$\chi(X) = \sum_\sigma (-1)^{dim(\sigma)}.$$

MAYBE TALK ABOUT TAME FUNCTIONS?

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

This results allows to compute in practice χ considering a finite triangulation of a manifold. As there is a homeomorphism between a manifold and it's triangulation χ will not change. We will be well advised to pick a tringulation with the least number of simplices to improve computational efficiency. For example the octahedron is a triangulation of a sphere. We will omit the process by which this is done in detail and refer the reader to [].

1.5.2 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 important 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.

The theory of Homology comes in two flavours - **simplical** and **singular**. Singular homology is geared towards analysing simplical complexes while singular homology is it's generalisation for arbitrary topological spaces. In this dissertation we restrict attention on singular homology because we are primarily interested in the computation of homology on simplical complexes. More information on singular homology can be found in the following sources [5, 4]

Homology is built around the interplay between two key concepts of **cycles** and **boundaries**. Let us consider the simplical complex on fig[] as an example. It consists of four vertices $\{a, b, c, d\}$, five edges $\{ab, bc, ca, db, cb\}$ and one face $\{abc\}$. 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. In our example we can observe that the edges ab, bc, ca and bd, dc, cb form a 1-dim cycle. This also happens to be 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

Notice also that the paths formed by the edges bc, ca, ab and ca, ab, bc are also cycles. The only difference is which vertex they start and end at. We would like to disregard the choice of starting point completely because those three paths represent the same

structure in the simplical complex and. To this end we shall introduce additive algebraic notation. In this notation the same cycle would be written as $ab + bc + ca$. We will soon demonstrate that additive notation is not only used to illustrate 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.

Talk about how the face covers a boundary. Talk about the problem of finding cycles that are not closed by boundaries. Those are exactly the holes in the space.

Let us be more formal now. To begin with, for simplicity, 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. The building blocks of the homology of a simplical complex X are:

- The vector space of **n-chains** of X. This is denoted as $C_n(X)$. It is an abstract vector space with basis all the n-simplices of X.
- The **boundary maps** of the n-chains of X. These are linear maps between the n-chains denoted as $\partial_n : C_n(X) \to C_{n-1}(X)$.

Now let us elaborate on these definitions. In our previous example $C_0(X)$ is the vector space that is spanned by the vertices $\{a, b, c, d\}$. 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 $\sigma \in C_0(X)$, then $\sigma = \alpha_0 a + \alpha_1 b + \alpha_2 c + \alpha_3 d$ where $\alpha_i \in \{0, 1\}$ for every $i = 0, 1, 2, 3$. If we go 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 linear combination $0ab + 1bc + 0ca + 1cd + 1bd$ and has coordinates $(0, 1, 0, 1, 1)$ in $C_1(X)$ with respect to the basis we have chosen.

We may of course wish to work with to use a different basis for some of the n-chains. Change of basis is useful in linear algebra and can have an effect on the computational efficiency, especially when dealing with projections and quotient spaces. This is completely acceptable in this setting and we can use any linear combinations of the simplicies so long as we obtain a number of linearly independent vector equal to the number of n-simplicies in the simplical complex or the dimension of $C_0(X)$. 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.

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 basis element $\sigma \in C_n(X)$ is that it returns the linear combination consisting of basis elements of $C_{n-1}(X)$ that are codimension-1 faces of σ. If σ is the affine combination of the vertices $[v_0, v_1, ..., v_n]$ the we define it's boundary as:

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

Linear functions commute with vector addition and scalar multiplication. This allows us to know everything that is to know about a linear function through it's effect on the basis vectors of its domain. This is because in the general setting for a linear function $f : V \to W$ we have that: $f\big(\sum_i a_i v_i\big) = \sum_i a_i f(v_i)$. We have demonstrated the effects of ∂_n on the basis vectors of $C_n(X)$. All we have to do is extend it linearly. This results in the following definition:

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

What we have thus obtained is a collection of vector spaces together with linear maps beteen. This is what called a chain complex. This is the quiver representation of a chain complex of a simplicat comples of dimension n.

$$C_n(X) \longrightarrow C_{n-1}(X) \longrightarrow ... \longrightarrow C_1(X) \xrightarrow{f_1} C_0(X)$$

For conveninece we will extend this chain on both sided with the zero dimentional vector space as follows:

$$0 \xrightarrow{\partial_{n+1}} C_n(X) \xrightarrow{\partial_n} C_{n-1}(X) \xrightarrow{\partial_{n-1}} ... \longrightarrow C_1(X) \xrightarrow{\partial_1} C_0(X) \xrightarrow{\partial_0} 0.$$

In this sequence ∂_{n+1} and ∂_0 are zero maps. In the case of ∂_{n+1} it maps the zero vector of 0 to the zero vector of $C_n(X)$ and ∂_0 maps all vectors in $C_0(X)$ to the zero vector in 0.

In this context we would like a simple method of recognising cyles and boundaries. The boundaries are provided to use by the boundary maps. Thus the set of all boundaries in $C_n(X)$ is given by the image of $C_{n+1}(X)$ under ∂_{n+1} or $im(\partial_{n+1})$. The cycles in $C_n(X)$ are given by all the vector in $C_n(X)$ that go to the zero vector of $C_{n-1}(X)$ under ∂_n. Intuitively the boundary of an n-chain is zero exactly when all of the faces of the simplices in the n-chain cancel out. The set of all vectors that go to the zero vector under the boundary map ∂ is called the kernel of ∂ or $ker(\partial)$.

From linear algebra we know that for a linear function $f : V \to W$ $ker(f)$ is a linear subspace of V and $im(f)$ is a subspace of W. In the context of chain complexes this means that the images and kernels of all the boudary maps are linear subspaces of their appropriate n-chains. What we would like ot learn about is of the relation of the cycles and boundaries in this algebraic context. We have defined the boundary operator in a

very special way. We have defined it so that whenever we apply it two consecutive times we obtain the 0 vector.

Lemma 2. *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 [5] for a more complete version.

Let us consider the boundary of σ 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 those will cancel out in the second boundary operation. □

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. □

Now, how do we only take into account the cycles that are not covered by a boundary? Take the quotient of the two. Because of the way how the quotient space is constructed the only cycles that do not go to the zero coset under the quotient projection map are the ones that entirely linear combinations of boundaries of higher dimensional simplicies. This is precicely how we define the n-th homology of a chain map.

Definition 19. *The n-th homology group of a chain map is* $H_n(X) = ker(\partial_{n+1})/im(\partial_n)$.

From linear algebra [1] 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. 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. Therefore $H_n(X)$ is a vector space and $dim(H_n(X)) = dim(ker(\partial_{n+1})) - dim(im(\partial_n))$.

Elements of the homology groups are called homology classes.

I will now give you some example of homology computations.

Now that we have ventured into the algebra and computed something based on the topology of the space it is time to interpret those results. What we are most interested is the dimension of the homology groups. The dimension of a finite dimensional vector space is the number of vector in a basis of that vector space. Thus if $H_n(X) \simeq \mathbb{Z}_2^m$ then $dim(H_n(X)) = m$. The dimensions of the homology groups are also known as the Betti numbers. The Betti numbers have the following topological interpretation.

- Betti zero - b_0 is the number of connected components
- Betti one - b_1 is the number one dimentional holes in a space or holes.

- Betti two - b_3 is the number two dimentional holes in a space or voids.

The higher order Betti numbers represent the number of higher dimensional holes. Given a nice enough topological space we can expect 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.

Give example with the torus

This is exactly what we wanted from Homology. An apparatus that allows us distinguish topological spaces based on the connectivity of their n-dimensional simplical complexes.

Before going forward we must note that we did not have to use coeficients in $\mathbb{Z}_2$ we could have equally used coeficients in $\mathbb{Z}$ but $\mathbb{Z}$ is not a field and we would have obtained that the $C_n(X)$ and $H_n(X)$ are not vector spaces but free abelian groups. If instead we had picked any arbitrary ring we would have obtained free modules instead of free abelian groups. We did indeed lose some information but sticking to vector spaces. The Betti numbers are not always equal, but by the Coeficient Theorem they are for suitably nice spaces. We readily refer the reader to [5] to learn about those. We shall continue the treatment of the subject in the same spirit of vector spaces.

There are two more notions we need to define to be able to fully utilise the power of homology - that of 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 topological space that consists of a single point. In that topological space all homology groups except for the H_0 are trivial. It is convenient in many application to make H_0 behave like the rest of the homology group and specifically be trivial in our example. In this sense path-connected topological spaces will have trivial zeroth homology. The geometrical interpretation of this extension is the reduces 0th homology counts the number of voids that separate path connected components of a topological space.

In formal terms we augment the chain complex of a topological space X with one additional group $\mathbb{Z}$.

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

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 reduces homology as the homology of the augmented chain complex or $\widetilde{H}_n(X)$. 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 useful notion is that of relative homology. Just as in abstract algebra we can

extract useful properties of quotient groups a natural question to ask is whether we can do something similar in the setting of algebraic topology and quotient spaces.

Let A be a subcomplex of X. Let us then define $C_n(X, A) = C_n(X)/C_n(A)$ and define a new relative chain complex as

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

This is a chain complex because...

More importantly the relative homology $H_n(X, A)$ is not defined as $H_n(X)/H_n(A)$ but as the homology of the chain complex defined above. As the boundary maps take $C_n(A)$ to $C_{n-1}(A)$ the boundary maps induce relative boundary maps on the chain complex. We call the cycles and boundaries of the relative chain complex relative chains and relative boundaries.

Intuitively here is how we can think of the relative homology classes [5].

A relative chain α is a relative cycle when it's boundary $\partial\alpha$ is in $C_n(A)$.

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

There is a way to relate the this purely algebraic machinery to our geometric intuition for "nice" enough spaces.
Theorem 2. *Excision - When A is*

A corollary of this is that if A is a close subcomplex of X then $H_n(X, A) \simeq \widetilde{H}_n(X/A, A/A)$ where A/A is a single point in X/A. This allows us to leverage our geometric intuition about quotient space to compute homology groups.

1.6 Differential Topology

Differential topology is the study of differentiable function on differentiable manifolds. As opposed to the theory of differential geometry, topologists are predominantly interested in the global structure of manifolds and primarily disregard local information such as for example curvature because it can be of little value in investigating the global properties of a space such as its genus. The classical example that illustrates this is that of the doughnut and the coffee mug. From the point of view of differential geometry they are not the same. You cannot rotate and translate one of them to obtain in other. From the differential topology perspective they are indistinguishable in that they share all the global properties topology can "see".

1.6.1 Morse Theory

One of the leading fields of differential topology is that of Morse Theory. It has shown to be fruitful in both theoretical investigations and real world application and computation []. Morse theory is the study of the deep relation between spaces and function defined on them. One of the main goals is to determine the shape of a shape by analysing a functions that can be defined on it.

Consider for example the real line $\mathbb{R}$ and the circle S^1. There are differentiable 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.

This example demonstrates the general methodology of Morse Theory. One defines a function on a smooth manifold and studies the critical points of that function as means of learning about the "shape" of the manifold. The class if real valued function on a manifold is far too complex though. This is why Morse Theory restricts it's attention to Morse functions.

Definition 20. *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.*

Based on this definition we will defined level sets, sublevel sets and superlever sets of a Morse function.

Definition 21. *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 interval $[-\infty, a]$ 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 $[a, -\infty]$. We shall call the path-connected components of a level set contours.

These sub(super) level sets allows us to decompose the manifold and work with chunks of it at a time.

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

- None of the critical points are degenerate.

- In Morse functions topological changes of the sub(super)level sets only happen at critical values.
- A Morse function defined on a close surface has a finite number of critical points.

1.6.2 Reeb Graph

The Reeb Graph is a tool that encapsulates the evolution of the 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 connected components 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 22. *Given a topological space X and a continuous function $f : X \to \mathbb{R}$ we can define an equivalence relation $\sim$ such that two points x, y in X are equivalent when there exists a path between them in a level set $f^{-1}(\{h\})$ for some $h \in \mathbb{R}$. The Reeb Graph is the quotient space $X/\sim$ together with the quotient topology.*

Intuitively it contracts all connected components to a single point.

SHOW LOTS OF PICTURES

The Reeb Graph is indeed homeomorphic to a topological graph. *Explain Why*

There are two important equations that [2] relate the Betti Numbers of the topological space X and the quotient space $X/_\sim$

$$\beta_0(X/_\sim) = \beta_0(X)$$
$$\beta_1(X/_\sim) \leq \beta_1(X)$$

.

We remind the reader that a contractable domain is on that is homotopic to single point. In the homology of such a space there is only one connected component and all other homology groups are trivial. Therefore $\beta_0 = 1$ and $\beta_n = 0, n \in 1, 2, ...$ A direct consequence of this is that the Reeb Graph of a contractable space is a tree regardless of the function defined on it.

1.6.3 Contour Tree

The special case of the Reeb Graph of a contractable space is called a Contour Tree.

Write about the algorithm for computing reeb graphs, say why they are slow

In order to move forward with the state of the art algorithms for computing the contour tree due to Carr, et. at []. We will define an additional equivalence relation on the quotient space that relates contour of different level sets.
Definition 23. *Let $C = \{p_0, p_1, ..., p_n\}$ be the set of all critical points where the number of connected components in the level sets changes.*
Definition 24. *Let α_1, α_2 be be two contours of two level sets (possibly the same level set). Then $\alpha_1 \sim \alpha_2$ when there exists a homotopy between α_1 and α_2 that does not pass through any point in C or when $\alpha_1 = \alpha_2$.*

Intuitively if we can deform one of the contours to the other along X without passing through a saddle then both contours correspond to the same edge in the contour tree. How about the contours that pass through critical points? They are only equivalent to themself because any homotopy to any other contour must pass through at least one critical point - the one contained in the contour.
Definition 25. *We will call contour classes with an infinite number of members infinite contour classes. Those correspond to the edges of the contour tree.*
Definition 26. *We will call contour classes with a single members finite contour classes. Those correspond to the vertices of the contour tree.*

Finite contour classes also correspond to closed sets of the form $[a, a] = \{a\}$ where a is the value of a critical point.

Infinite contour classes correspond to open sets of the form (a, b) where a and b are the values of critical points and where for any two $c, c' \in (a, b)$ there is a contour in $f^{-1}(\{c\})$ that is equivalent to a contour in $f^{-1}(\{c'\})$.

Leafs correspond to local minima or maxima.

Interior vertices are one where at least one infinite contour class is created and at least one infinite class is destroyed [?].

Show many fabulous pictures of contour trees

1.7 Algorithms for Computing Contour Trees

In this section we will present the state of the algorithm for construction a contour tree. * They are based on the works of [] and we will try to relay them in as much detail as

possible.

1.7.1 Input Data Format

UNDER CONSTRUCTION

In order to simplify out implementation without much loss in generality we will assume that the input data comes in the form of a real valued $n \times m$ matrix or more appropriately in our context grid. The gird represents the values of vertices of a two dimensional simplicial mesh where the vertices are "evenly spaced" measurements. See example []. The algorithms presented here can be extended to a grid of any number of dimensions or even to irregular grids. See [].

The underlying simplicial mesh is assumed to be a triangulation of the vertices [] and the values at the edges and faces of the mesh are obtained through piecewise-linear interpolation based solely on the values of the vertices. Our final assumption is that the input values in the grid are unique. This ensures that the interpolated function is Morse and the it's critical points lie in unique vertices.

If we are only computing the contour tree of the mesh and not bothering with extracting contours and level sets uniqueness of data values allows us to avoid having to compute the interpolation function and to only work with the values at the vertices of the mesh.

1.7.2 Height Trees

In order to discuss the algorithms for computing contour tree we must first establish some notation and useful properties about height graphs and trees. A height graph is a graph $G = (V, E)$ together with a real valued function h defined on the vertices of G. A height tree is unsurprisingly a height graph which is a tree. Contour trees are examples of height trees and in the spirit of the assumptions we have made about our input data in the previous subsection we will assume all vertices 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 27. *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 28. *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.*

Conversely if $\delta^+(v) = 0$ and $\delta^-(v) = 1$ then v is an upper leaf. We will see in the next chapter how we can use these two types of leaves to construct the contour tree.

1.7.3 Join and Split Trees

The contour tree can be associated with two other trees. Those are the join and split trees. They each contain one half of the topological information of the contour tree. 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 []. More formally 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 just as in the way sublevel and super level sets are - relative to the direction of travel of the interpolation function.

Every contour tree is associated with a unique pair of join and split trees. The core idea behind the algorithm for computing the contour tree is that the join and split trees can be combined together to produce the contour tree. The core observation that makes this possible is that the join and split trees of the mesh and of the contour tree of the mesh are the same. The algorithm that is proposed in[] leverages those two insights and has two phases. First it constructs the join and split trees of the mesh and then combines them to obtain the contour tree.

Let us now examine how join and split trees are compute. We will describe for the process for the join tree only as it is completely analogous in the split tree case.
Definition 29. *A join component is a connected component in the sublevel set $f^{-1}(\{h\})$ at some $h \in \mathbb{R}$.*

Let us now formalise the notion of tracking join components and constructing a join tree. Let us work in the general setting where X is any path-connected topological space and $h : X \to \mathbb{R}$ is a function defined on X. The claims we make will hold in the special case where X is a simplicial complex. Let us consider all sublevel sets $X_t = h^{-1}((-\infty, t]) = \{x \in X : h(x) \in (-\infty, t]\}$. They form a one parameter family $\{X_t\}_{t\in\mathbb{R}}$ of nested subsets where $X_a \subseteq X_b$ whenever $a \leq b$. What the join tree captures is how the connectivity of the sublevel sets changes as the parameter t is increased.

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. Furthermore if $X_m = X$ is the last sublevel set for some $m \in \mathbb{R}$ then all join components merge into one because X is path connected.

* Here is a beautiful example of this. *

In order to compute the join tree of our input mesh we will perform an upwards sweep on the vertices. We will use the union-find data structure [] to keep track of which join components vertices are in. Initially all vertices will be places in their unique connected component. At the end of the sweep they will all be in the same component. Then we perform an upwards sweep through the vertices and check if the current vertex merges two or more join components. If it does we add an edge between that vertex and the oldest vertex in the join components in merges in the merge tree.

Not all vertices of the mesh will be in the join tree. Only those which correspond to local maxima and to join saddles. This will pose a problem upon combining 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.

Note that the same augmentation can be applied to the contour tree as well.

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

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 30. *Property of leaves*

Once we have those we can remove them from the join and split trees via vertex contraction. Another remarkable we have is that after applying vertex contraction the new join and split are of the subgraph of the contour tree induced by all non-leaves. This means that we can obtain the so called 1nd order leaves (vertices of distance 1 from a leaf) from the new join and split trees by the first property and add those to the contour tree. We can repeat this process iteratively by "pruning" until we have no more vertices left in the join and split trees. Upon reaching this state the contour tree is fully computed.

* See example with pretty pictures *

1.7.4 Serial Algorithm

To summarise what we obtained so far, here is the overall algorithm for computing the contour tree.

Step 1. Read input grid and convert it to a triangulated mesh.

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

Step 3. Iteratively prune leaves and adjacent from the Join and Split tree and add them to the Contour Tree until they are empty.

Step 4. Remove augmentation if necessary and output contour tree.

The running time of this algorithm is $O(n\alpha(n))$ where alpha is the inverse Ackerman function. For all practical intents and purpose the running time of this algorithm is linear. It's space complexity is also linear.

1.7.5 Parallel Algorithm

How do we ensure the logarithmic collapse? Via theorem []. About half the vertices are leaves.

The parallel algorithm can process long monotone path on a single pass. Thus at every pass the number of vertices should halve. But this is not case because of the w-structures.

1.7.6 Contour Tree Simplification

Contour Trees are summary of the connectivity of the level sets of a Morse function. They are primarily used in scientific visualisation. A central problem in visualisation is simplifying the data that is presented 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 remove parts of the contour tree that correspond to less "significant" topological features.

One such technique is Branch Decomposition. It was first introduced in [9]. It involved decomposing the contour tree into a set of disjoint monotone paths which cover all edges of the tree. This is called branch decomposition and the trivial one is where we take every vertex to be a separate path. A more complex one is shown in this example. 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.

The branches in this scheme represent pair of critical points and form the basis for topological simplification that can be applied. We apply a simplification by removing a branch that does not disconnect the tree. This produces a hierarchy of cancellations like in example []. 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 [6]. In Chapter N of this dissertation we will demonstrate that this claim is either incorrect of misleading. Stay tuned folks.

1.8 Additional Proofs

Lemma 3. *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 4. *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.

□

Chapter 2

Something "W" This Way Comes!

We will now continue the discussion on the difficulties of parallelising the algorithm for the computation of the contour tree in a more formal setting. In this Chapter we will develop theory that captures the informal description we outlined previously. We will use this theory to construct three algorithms for the detection of the largest w-structure in a height tree. We will also provide pseudocode, proof of correctness and proof of the space and time complexity of all presented algorithms.

2.1 W-Paths in Height Graphs

What we are interested in are the paths in the height tree which form a zigzag pattern. As shown in fig[] they can be decomposed into monotone paths of alternating direction that share exactly one vertex. More formally, if P is a path in a height tree we can always decompose it into vertexwise maximal monotone subpaths $(P_1, P_2, ..., P_k)$ 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 $k \geq 1$.

One way to characterise paths in a height tree is by the number of subpaths in their monotone path decomposition. The maximum path with respect to this property is precisely the lower bound on the parallel algorithm introduced in []. As a special case we must note that paths that can be decomposed into less than four monotone paths do not pose an algorithmic problem. 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. 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 cumbersome notation in this context we shall adopt a slight abuse of notation and in the future write similar statements 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. Intuitively this is similar to how the length of a path measures the number of edges between it's vertices. We will make an analogous definition of the w-length of a path as the number of inside

vertices which are kinks. Let us denote a path from u to v as $u \rightsquigarrow v$ and with $d(u \rightsquigarrow v)$ measure the length of the longest path between u and v and with $w(u \rightsquigarrow v)$ the path with the largest number of kinks (or the longest w-path). One immediate observation we can make is that $w(u \rightsquigarrow v) < d(u \rightsquigarrow v)$ for any two vertices in any graph. This follows from that fact that the longest path between u and v also has the largest number of vertices. A path with k vertices has length $k-1$ and $k-2$ internal vertices which may or may not be kinks.

2.2 W-Paths in Height trees

We will not restrict our attention to height trees. Those are unsurprisingly height graph which are trees. The first key property of trees will make use of is that there is a unique path between any two vertices. This allows us to slightly simplify some of our notation. Instead of $d(u \rightsquigarrow v)$ and $w(u \rightsquigarrow v)$ we will write $d(u, v)$ and $w(u, v)$ respectively.

We are now fully prepared to unveil that which we seek - the longest w-path in a tree (the one with the most kinks). As there is a unique path between any two vertices this can be posed as an optimisation problem as follows:

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

The search space is quadratic in the number of vertices and this can be computed by running a modified version of Breadth First Search (BFS) from every vertex in the tree. This modified BFS computes the w-distances from a starting vertex to all others. The presudocode for this modification is presented in []. The running time for this is $O(n^2)$ and is far from satisfactory given that the actual algorithm for construction the contour tree runs in time $O(n\alpha(n))$. This is because in practical terms a $O(n^2)$ algorithm is completely unusable on datasets which a near linear time algorithm can process.

And indeed we can do better. As the reader may have noticed the definitions we have made so far are analogous to the task of computing the longest path between any two vertices of a tree. This is completely intentional as we will demonstrate how algorithms for computing the longest path in a tree can be modified to produce the longest w-path instead. Finding the longest path of a graph in the general case is an *NP-hard*. Fortunately the Contour Tree is a tree. The longest path in a tree is known in the literature as it's diameter and has a polynomial time algorithm. The two most popular linear time algorithms found in the literature I will denote as Double Breadth First Search (2xBFS) and Dynamic Programming (DP). We will now take a look at how these algorithms work and hint at how they can be adapted in the next chapter.

2.2.1 Double Breadth First Search

This algorithm works in two phases. First it picks any vertex in the tree, say s, and finds the one farthest from it using Breadth First Search (BFS). Let us call that vertex u. In the second phase it runs a second BFS from u and again records the farthest one from it. Let us call that v. The output of the algorithm is the pair of vertices (u, v) and the distance between them, $d(u, v)$. That distance is the diameter of the tree.

This algorithm has linear time complexity as it consists of two consecutive linear graph searches. It's correctness if a direct consequence of the following Lemma.
Lemma 5. *Let s be any vertex in a tree. Then the most distant vertex from s is an endpoint of a graph diameter.*

The proof of this Lemma can be found at []. In the next section we shall demonstrate how this proof can be modified to produce a near optimal algorithm linear time algorithm for finding a path whose w-length is bounded from bellow by the w-diameter of a tree.

2.2.2 Dynamic Programming

The second approach is based on the Dynamic Programming paradigm. It is most often applied to 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.

Naturally, 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 analogously. In a rooted tree let v and u be two vertices such that v is the parent and u is the child. 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. Clearly 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. This definition will allows us to recursively consider all subtrees of a rooted tree $\{T_u\}_{u \in V(G)}$. Also note that if all vertices in T_u except u inherit their parent from T then T_u is also a rooted subtree - it's root being u.

To continue we will need to define two functions on the vertices T. 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 $D(u)$ longest path in T_u. The function we will maximize is $D(s)$ where s is the root of T. With these two function we are now ready to recursively define the value of the optimal solution:

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

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. We are guaranteed to reach the base case as each subtree is strictly smaller and we bottom out at the leaves.

This algorithm can be implemented in linear time through Depth First Search (DFS) by using two auxilary arrays that hold the values for $h(u)$ and $D(u)$ for every $u \in V(T)$. We will omit a formal proof of correctness and refer the reader to []. The proof relies on the fact that the longest path in a rooted tree either passes through the root and is entirely contained in the subtrees rooted the children of the root.

2.3 W Diameter Detector

We will now step into the realm of w-detection. Before we outline the proposed algorithms we must establish two key properties which hold the difference between the tree diameter algorithms and their modification to tree w-diameter algorithms.

Definition 31. *Subpath Property*

Let $a \rightsquigarrow b$ be a path and $c \rightsquigarrow d$ it's 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 for the w-length, decreases only when we remove a kink from a path.

Definition 32. *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:

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

Indeed 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 counted by either $w(a, u_i)$ or $w(u_i, b)$. When making use of path decomposition property in future proofs we must account for whether the vertex we are decomposing a path at is a kink in that path or not.

2.3.1 Linear Time Algorithm - 2xBFS

We shall first explore how we can modify the Double Breadth First Search algorithm to compute the w-diameter of a height tree. The new algorithm will follow exactly the same steps. The only exception is that it will run a modified version of BFS that computes w-distances [see algorithm next page] from a given root vertex to all others in the tree. The algorithm works by first running a BFS from any vertex in the graph and then records the leaf that is farthest in terms of w-length. This furthest leaf is guaranteed to be either the endpoint of a path in the tree whose w-length least that of the actual w-diameter of the tree minus two.

Before proving the correctness of the algorithm we must first establish two useful properties that relate the w-length of a path to it's subpaths.

Lemma 6. *The Algorithm produces the endpoints of a path who is at most 2 kinks shy of being the kinkiest path in the tree.*

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:

$$w(s, u) \geq w(s, t), \forall t \in V(T) \tag{2.1}$$

$$w(u, v) \geq w(u, t), \forall t \in V(T) \tag{2.2}$$

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:

$$w(a, b) \geq w(c, d), \forall c, d \in V(T) \tag{2.3}$$

By this equation we have that $w(a, b) \geq w(u, v)$. Our goal in this proof will be to give a

Algorithm 1 Computing the W Diameter of a Height Tree.

```
1: function W_BFS(T, root)
2:     root.d = 0
3:     root.π = root
4:     furthest = root
5:     Q = ∅
6:     Enqueue(Q, root)
7:     while Q ≠ ∅ do
8:         u = Dequeue(Q)
9:         if u.d ≥ furthest.d then
10:            furthest = u
11:        for all v ∈ T.Adj[u] do
12:            if v.π == ∅ then
13:                v.π = u
14:                if h(u) ∉ (h(v), h(u.π)) then
15:                    v.d = u.d + 1
16:                else
17:                    v.d = u.d
18:                Enqueue(Q, v)
19:    Return furthest
20: function CALCULATE_W_DIAMETER(T)
21:    s = <any vertex>
22:    u = W_BFS(T, s)
23:    v = W_BFS(T, u)
24:    return v.d
```

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 2.2 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:

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

But as $w(u, v) \geq w(u, b)$ (by equation 2.2) we obtain that:

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

Considering all possible values that x and y can take, we can see that the minimum value for the right hand side of the inequality is at $y = 0$ and $x = 1$. 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 2.1 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 2.2)we obtain that:

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

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

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

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

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

□

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

Proof. The modified BFS function has the same time complexity of BFS as all we have added 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 twice remains in linear, thus the overall complexity of the algorithm is linear as well. □

Lemma 8. *The space complexity of the algorithm is $O(|V|)$.*

Proof. Completely analogous to the standard BFS algorithm, this algorithm uses the same amount of memory in the standard memory model. □

2.3.2 Pathological Cases in 2xBFS

Describe all of them

Sexiest Greek letter?

$$\xi$$

$$\zeta$$

$$\chi$$

$$\nu$$

2.3.3 Attempts at resolving the accuracy of 2xBFS

For the intents of purpose of this dissertation the accuracy of this algorithm is sufficient. In large enough data sets this estimate provides enough insight to correlate the observed iterations needed to collapse the split and join trees and the resulting w-diameter of the tree. This is demonstrated empirically in Chapter 3. Regardless of such practical considerations it is still of inherent theoretical interest to investigate how we may be able to obtain a more accurate modification of this algorithm.

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 fact in some cases it is possible that each successive BFS will return the same path over and over again. Obverse how in @TODO fig[] 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 points and keep the maximum value found. This approach is somewhat reliable, but may still fail to find the w-diameter. Consider @TODO fig[]. That artificial example shows that there can simply be too few starting points which would produce the w-diameter.

In the search for a better solution let s be a starting vertex and let the vertices

$U = \{u_1, u_2, ..., u_n\}$ be the furthest away in terms of w-distance and
$W = \{w_1, w_2, ..., w_n\}$ be the ones second furthest away. By the proof of the algorithm we know that not necessarily all vertices in those sets would produce a w-diameter. Thus lets us define $R \subseteq U \cup W$ as the set of vertices which are endpoints of a w-diameter. As we have shown we can construct an example where $|R| = 2$ and $|U \cup W|$ is arbitrarily large. Can we then find some property of the vertices in R and pick them out in the first phase of the algorithm? *This I will leave open for the future generations to ponder. I hope in doing so all people of the world will unite unite and end all wars and prejudices in order to work towards this common good!*

2.3.4 Dynamic Programming Algorithm - DP

Idea redefine $N(u) = N(u)/u.\pi$ so you can simplify notation.

While it is encouraging that we have obtained an algorithm that bounds the w-diameter it is also quite 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.

This may seem like a simple substitution at first glance, but the devil is in the details. As in the previous modification all additional difficulties stem from the difference in combining path lengths and path w-lengths. Let us begin by examining how the w-height of a vertex is computed from the w-heights of its children. Let s be a vertex in T and let us assume the we have computed the w-heights of its children recursively. In the case of computing the height we can simply set $h(s) = \max_{u \in N(s)} \big(h(u)\big) + 1$. 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 $w(u) = \max_{v \in N(s)} (w(v))$. 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.

To see how we can 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. Then we can compute the w-height of s as:

$w(s) = \max_{u \in N(s)} \{h(u) + \max_{v \in L_u}(w_{s,v}(u))\}$. 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. There is no point in looking at paths of lesser w-length as it can only increase by one and at best match the maximum ones.

In the tree diameter scenario path combination is straightforward. For a tree with root s 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 get a walk and not a path). Next we will combine them to obtain the longest path that goes through s. This path combination yield the sum $h(u) + h(v) + 2$, where we account for the two additional edges $us, sv \in E(T_s)$. This reasoning of course extends to all subtrees in T. In the latter 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 the tree 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-length of two paths - one 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 can 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.

This process is similar to the one for obtaining the w-height of a vertex and is described by the following formula:

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

Thus the formula that describes the optimal solution can be written as:

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

As before the optimal solution is either entirely in one of the subtrees of the children of a vertex or in the path combination of two of the children of the vertex.

Which one looks better?

$$W(s) = max \left\{ \begin{array}{l} \max\limits_{u \in N(s)} \Big(W(u) \Big), \\ \max\limits_{u,v \in N(s)} \Big(\max\limits_{u' \in N(u)} \Big(w(u') + w_{u',s}(u) \Big) + \max\limits_{v' \in N(v)} \Big(w(v') + w_{v',s}(v) \Big) + w_{u,v}(s) \Big) \end{array} \right\}$$

Algorithm 2 Computing the W Diameter of a Height Tree.

```
* NEW CODE I GYNORMOUS SHOULD I PUT IT HERE *?
 1: function W_DFS(T, s)
 2:     if |T.Adj[s]| == 1 AND s.π ≠ s then
 3:         s.W = 0
 4:         s.w = 0
 5:         return
 6:     for all u ∈ T.Adj[s] do
 7:         if u.π == ∅ then
 8:             u.π = s
 9:             W_DFS(T, u)
10:
11:     Array p
12:     for all u ∈ T.Adj[s]/s.π do
13:         p[u] = 0
14:         for all v ∈ T.Adj[u]/u do
15:             p[u] = max(p[u], v.h + w_{v,s}(u))
16:     maxCombine = 0
17:     for all u ∈ T.Adj[s]/s.π do
18:         for all v ∈ T.Adj[s]/s.π do
19:             maxCombine = max(maxCombine, p[u] + p[v] + w_{u,v}(s))
20:
21:     maxSubsolution = 0
22:     for all u ∈ T.Adj[s] do
23:         maxSubsolution = max(maxSubsolution, u.W)
24:
25:     s.W = max(s.maxCombine, s.maxSubsolution)
26: function CALCULATE_W_DIAMETER(T)
27:     s = <any vertex>
28:     s.π = s
29:     W_DFS(T, s)
30:     return s.W
```

Lemma 9. *The Algorithm produces the w-diameter of a height tree.*

Proof. TBA □

Time for the proof of correctness.

Time for the proof of correctness.

Proposition 2. *Given a rooted tree T the w-diameter of T either passes through the root or is entirely contained in one of the subtrees of the children of the root.*

Proof. This is trivially true there is simply nowhere else it can be. □

Therefore the optimal solution is obtain either through one of the optimal solutions of the children or through path combination. All we have to do is show that path combination produces the longest w-path that goes through the root of a subtree. The rest will follow from the prop[]. It is the same as the tree diameter algorithm.

Proposition 3. *The combine path subroutine compute the correct answer.*

Proof. This is pretty obvious. We are using maximum path and maximising the oportunities for kinks. If there are two maximum paths all with kinks we will detect them. There cannot be a kinkier path there is simply nowhere it could be as it has to pass through the root and two of it's children. □

As path combination is correct then the optimal sumproblem function is correct. Then the whole algorithm must be correct.

The complexity of the proposed solution is:

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

Where $\sum_{u \in V} \sum_{v \in N(u)} d(v)$ is the loop over all children of children and $\sum_{u \in V} d(u)^2$ is the double loop over all children in the final path combination.

Firstly we can show that:

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

This is because as we are in tree, every vertex will be visited exactly once as a child of a child. If it were visited twice then there would be two distinct paths to that vertex which would mean a cycle.

The other argument is more difficult to bound. One thing that is clear is that

$$\sum_{u \in V} d(u)^2 \geq \sum_{u \in V} d(u) = 2|E|$$

This is true because the degree of a vertex is a positive integer and for any $x \in Z^+, x^2 \geq x$. This lower bound shows that it may be possible to obtain linear time

complexity. I will demonstrate how we can bound it from above.

A triangle is the complete graph on three vertices. As trees have no cycles they cannot have induced triangles. Therefore for any edge in a tree $uv \in E(T)$ we have that $d(u) + d(v) \leq |V|$. Indeed, if we do not have an induced triangle there are no vertices that $d(u)$ and $d(v)$ count twice. Summing over all edges we get that:

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

The key to solving this is to notice is that if we expand the summation every term $d(u)$ will be present exactly $d(u)$ times (one for each of it's edges). This allows us to obtain that:

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

Overall for the two sums we have shown that:

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

Therefore the time complexity of the dynamic programming solution is:

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

The running time is quadratic. Theoretically this is no better than a brute force exhaustive search. Despite this we have reasons to believe that it has the potential for better practical performance. The main reason that leads us to this conclusion is that the quadratic behaviour comes from the double loop on the children of all vertices. We know from the **lemma in previous chapter** that in any tree for any vertex of degree d there are at least d distinct leaves. Therefore for any vertex of high degree there will be as many vertices which are base cases for the recursion and will take constant processing time. This behaviour is/is not demonstrated in the next chapter where implementations of both w-diameter algorithms are compared empirically.

[8]

Chapter 3

Empirical Study

Whole Chapter is Under construction

The main objective of the empirical study if twofold. Firstly to correlate the iterations needed to collapse the contour tree in the merge phase of the algorithm with its w-diameter. Secondly to demonstrate that w-structures are found in real world data and do appear more abundantly as the size of datasets increases. As a result of this study we will be able to shed light on the practical obstacles that the w-structures pose. We are hopeful that this is the first step to overcoming them.

In addition to the main objectives there are some additional topics, related to the empirical study, that will be discussed:

- Description the implementation of all used algorithms.
- Demonstration of the running time of the implementations of the algorithms.
- Exploring the statistical distribution of w-structures in randomly generated trees.
- Correlating itterations for collapse with w-diameter of randomly generated tree.
- Determining the smallest 2-dimensional grid dataset that exhibits a w-structure.
- Discussion on whether it is possible to determine the w-diameter of a datasets without computing it's contour tree beforehand.
- Discussion on the kinds of structures in raw data that produce w-structures in the contour trees.

3.1 Implemented Algorithms

For the dissertation I implemented the following algorithms - contour tree, 2xBFS, DP.

The first algorithm that had to be implemented was for computing the Contour Tree. For simplicity I implemented a version that takes as input a rectangular 2-dimensional mesh. I used the pseudocode provided in this paper [] and implemented it in C++. To test the correctness of the code I compared it against the code that was used for [].

After this I implemented three algorithm for w-detection. The first one is the most basic exhaustive approach where we find the w-length of all paths in the tree. It has quadratic

running time. The second one is the 2xBFS algorithm and lastly the DP algorithm. They were implemented in C++. You can find the code in the appendix.

To implement the DP algorithm I had to modify to recursive solution to a bottom up solution.

Another algorithm I implemented is one that generates all possible $n \times m$ 2-d grids. The objective here is to find the minimum one that produces a w-structure.

Lastly I implemented the algorithm in [] that computes extended persistence. I did this to verify the claims that I will make in the next chapter about the pairing of the global minimum and maximum.

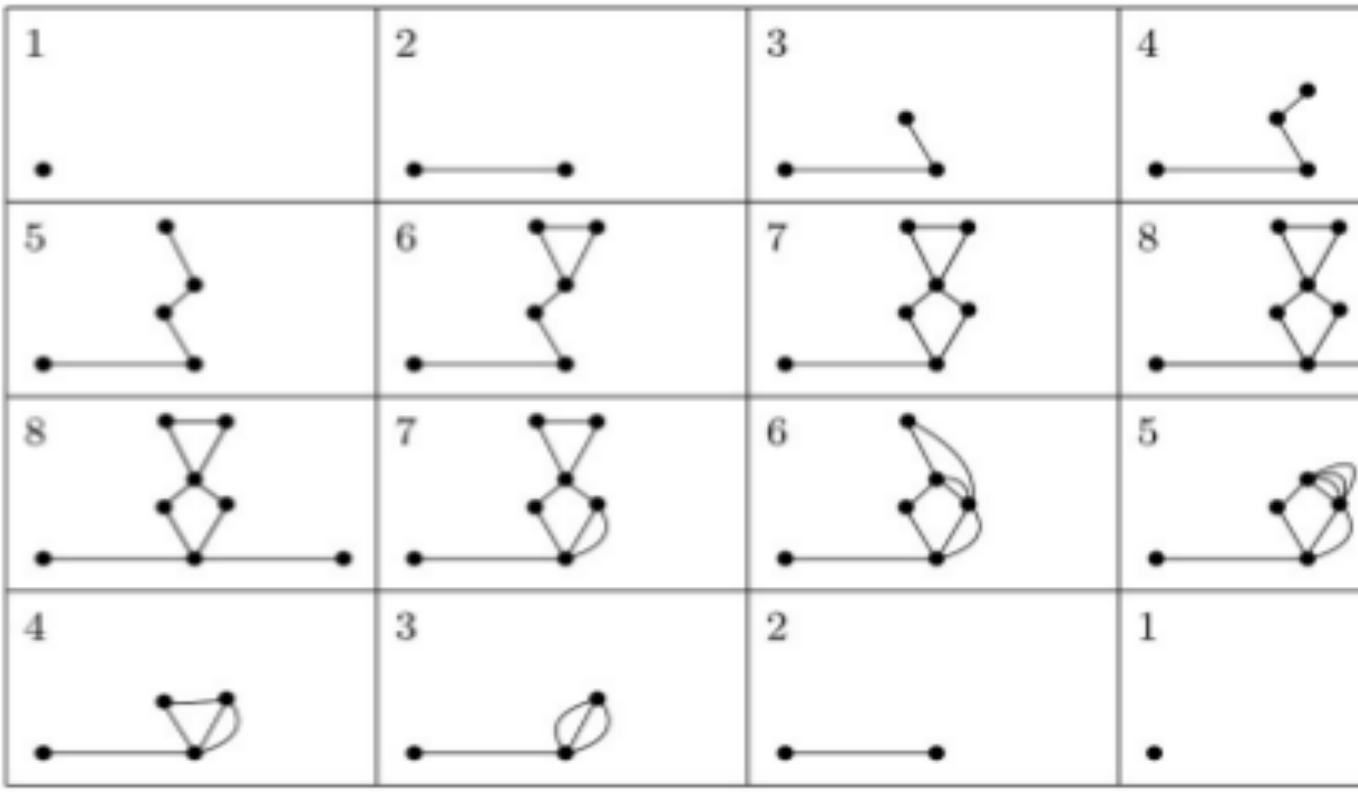
All of the implementation are serial and not attempt has been made to parallelise them. There was simply no point in that as it is not the main objective of this dissertation.

I have also used several third party applications. The first one is Hamish Carr's serial and parallel implementations of the CT. I have also used software that computer persistent homology called Perseus* to test the correctness of the implementation of the one I have.

3.1.1 Running Times

Now we will demonstrate that the running times of the algorithms we have implemented correspond to the theoretical results we proved in the previous chapter. We will only test the 2xBFS and DP algorithms because they are the only novel ones. As such they have not yet been implemented to out knowledge. The biggest contribution here is demonstrating is that the DP algorithm does scale linearly with randomly generated data input. This shows that the average running time of that algorithm may be linear and not quadratic which it's worse case running time. A more detailed further algorithmic analysis is needed to prove this theoretically, which we will not do.

The following chart shows the running time of the two algorithms on a sequence of randomly generated trees. The number of vertices in the trees is plotted on the horizontal axis and the running time in seconds is plotted in the vertical axis.



This graph shows that the running time of the two algorithm is linear. This is further supported by the statistical analysis through linear regression. We have fit a line through the points with *figure out how to do this.*

3.2 Analysing Datasets

I will analyse three types of datasets. Two of the types are taken from real life data and one is synthetic. The first type is data that is the elevation of a mountain range in Canada. The second one is images. The third one is randomly generated graphs. The goal here is firstly to demonstrate where w-structures appear in the contour trees of real life data. The second goal is to analyse random graphs and derive statistical information on the probability of having large w-structures in contour trees of large data sets.

This table is for augmented contour trees.

3.2.1 Mountain Range Data

This is elevation data taken from the Canadian Mountains. Oh Canada is so great and amazing. Oh motherland of maple and Celine Dione I bow before you beautiful nature and spectacular mountains.

The datasets are all part of something. Explain where they are from.

Dataset	Vectices	2BFS	DP	NBFS	Diameter	Iterations
vanc	378	2	2	2	311	-1
vancouverNE	4851	4	5	5	1338	-1
vancouverNW	4900	5	5	5	1456	-1
vancouverSE	4950	6	6	6	1306	-1
vancouverSW	5000	4	4	4	1977	-1
vancouverSWNE	1250	5	5	5	423	-1
vancouverSWNW	1250	3	3	3	712	-1
vancouverSWSE	1275	3	3	3	759	-1
vancouverSWSW	1225	2	2	2	845	-1
icefield	57600	7	7	7	12280	-1
pukaskwa	180	180	182	-1	374866	-1
gtopo30w020n40	-1	-1	-1	-1	N/A	-1

All the vancouver data sets are similar. There we have a very low w-diameter compared to diameter. Speculate as to why that may be the case.

The most interesting case is pukaskwa. Notice how pukaskwa has 881600 edges this means that under logarithmic collapse we should take 13 iteration. Instead we do 90. This is a problem, no?

3.2.2 Images

Find some images and test them and write about them.

3.2.3 Random Data

Do random trees to get the distribution of Ws. Do random trees to get the iterations correlation of Ws.

Talk about the value of random data in providing statistics. It may not be realistic but we may draw conclusion about the distribution of w-diameters in random trees.

This is taken from generating random data sets and taking the distribution of the w-diameters of the trees. As you can see it kind of looks like a normal distribution. Interesting is it not? Talk about random samples and the law of large numbers.

Here the overall conclusion that can be obtained from this analysis.

3.2.4 Conclusions

These are the conclusions from the empirical study.

- 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.

Also find some data sets to analyse. Maybe do some medical 3-dimentional data sets.

Talk about why random data sets may not be completely reliable.

Show some graphs and shit

Make some reflective summary of scheize

3.3 Finding the smallest W-structure

An interesting question that arises is what is the smallest dataset that produces a w-structure of at least three kinks. This has educational value. It's also useful for out general understanding. It will also serve as a very important counterexample in the next chapter. It is good for counterexamples to be as small as possible. That way they it's easier to articulate the counterarguments.

3.4 Getting the w-diameter from raw data

This analysis was all well and good, but it doesn't do too much good as it is done after the contour tree has been computed. The next step is to produce and algorithm that either produces it from raw data or produces it from the join and split tree. The hope for this would that is there is some priori information before going into the merge phase of he algorithm, we may be able to avoid the serialisation along the kinky paths.

3.5 Future work for the empirical study.

Summarise things say what was successful, what was not. That kind of stuff.

This chapter does seem short. This is because most of the work put in the dissertation has either been theoretical which is in the previous chapters. or on impelmenting the newly created algorithms, which are in the appendix.

Chapter 4

Persistent Homology and Contour Trees

We will now take a look at one of the tools that has made topological data analysis so viable in the recent years. This tool is called Persistent Homology (PH). In this chapter we will first further develop the mathematical framework of Homology to accommodate this new concept. Following our theoretical foray we will examine the practical aspects of the computation of persistent homology and its relation to the computation and simplification of contour trees.

4.1 Induced Maps on Homology

Before introducing ourselves with persistent homology we will take a slight detour in order to introduce the last piece that we are missing to enable its construction. There is a general result in singular homology that shows the interaction of continuous maps and homomorphisms between homology groups.

Definition 33. *Let X and Y be two topological spaces. Let $f : X \to Y$ be a continuous function. Then f induces a homomorphism $f_* : H_n(X) \to H_n(Y)$ for all $n \in \{0, 1, 2, ...\}$.*

This means that if we have a continuous function between two spaces we can immediately associate the homology classes of X to those of Y. All we have to do to obtain the induced map is to compose the simplices with the continuous function f. The details of this process are outlines in [5].

This general result is not appropriate for simplicial complexes. WHY?! We need a more tracktable definition to aid us in our computation. We will thus present the following combinatorially flavoured definition given by [7].

Definition 34. *Let X and Y be two finite abstract simplicial complexes. A function $f : X \to Y$ is a simplical map when if σ is a simplex of X then $f(\sigma)$ is a simplex of Y.*

The two most important observations we can make based on this defitions are the following:

- The composition of two simplicial maps is simplicial.
- When Y is a subcompex of X the inclusion map is a simplicial map.

The reason why we introduced simplicial maps is so that we can pose the following question. If there is a simplical map between two simplicial complexes, can we use it to

relate their homology classes? The answer is yes, we can thanks to [7]!

Definition 35. *Let X and Y be two simplical complexes and $f : X \to Y$ be a simplicial map. Then f induces a homomorphism $f_* : H_n(X) \to H_n(Y)$ for all $n \in \{0, 1, 2, ...\}$.*

The homomorphism is induces by taking the simplicies of a chain through the simplicial map and the considering the homology class the chain ends up in (if any). Detail on this can be found in [7].

We will further expand this definition to also cover relative chain maps and relative homologies.

Definition 36. *Let X and Y be two simplical complexes and let $A \subseteq X$ and $B \subseteq Y$ be two subcomplexes. Let $f : X \to Y$ be a simplicial map such that $f(A) \subseteq B$. Then f induces a homomorphism $f_* : H_n(X, A) \to H_n(Y, B)$ for all $n \in \{0, 1, 2, ...\}$.*

We will use the shorthand $f : (X, A) \to (Y, B)$ for functions that satisfy the criteria of this definition. The function f is called this a simplicial map between simplicial pairs (analogous to continuous map between topological pairs in [5]).

The homomorphism is induced by running the relative homology classes through the simplicial map and recording which class their image lands in. The primary type of map we will use in this chapter is a specific kind of simplicial map - the inclusion map. The reason for this will become clear in the following section. In the case of absolute homology when X is a simplicial complex and A is a subcomplex of X there is a natural inclusion map $i : A \to X$ which is injective but not necessarily surjective. It takes the simplicies of A to exactly the same simplicies of X and leaves the simplicies outside of A untouched.

We shall define the inclusion of relative homology analogously. Let B be another subcomplex of X such that A is also a subcomplex of B, or $A \subseteq B \subseteq X$. Then let $i : X \to X$ be the identity map. As $A \subseteq B$ then the restriction $i_A : A \to B$ is a well defined function and therefore $i(A) \subseteq B$. Therefore there is a map i between the pairs (X, A) and (X, B) such that $i(A) \subseteq B$ by the previous definition this map induces a homomorphism $i_* : H_n(X, A) \to H_n(X, B)$.

4.2 Persistent Homology

Persistent Homology emerged in the early 2000s in the this work of [6]. The original motivation for introducing it was to better model point cloud data through filtrations of Vietoris Ribs complexes. Persistent Homology has since grown into a general methodology that can be applied any filtration of a topological space. To best illustrate what persistent homology is let us consider a filtration of a simplical complex X. Example in fig[].

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

We have obtained a one parameter sequence of nester subcomplexes. Another way to think of this 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 in time. We can already compute the homology groups of the individual X_i. The key insight in persistent homology was to as the question whether we can track the evolution of individual homology classes in the homology groups as we go from one complex to the next. This is made possible by the subset relation between all of the X_i. As discussed in the previous section the inclusion map is the natural map between a set and it's superset. More formally we have inclusion maps $i_{i,j} : X_i \to X_j$ for $i \leq j$ because $X_i \subseteq X_{i+1} \subseteq ... \subseteq X_j$. By only considering the inclusion maps between consecutive X_i and X_{i+1} we can build the following chain of simplicial complexes

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

where we have renamed all inclusion maps to i and infer them from context. We have already shown that the inclusion maps are simplical and that simplical maps induce homomorphisms on homology groups through chain maps inducing. This lets us transform the sequence directly to the homology groups like so:

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

Here it is important to note that the induced maps i_* do not have to be the inclusion maps on the homology groups. They can easily fail to be injective when for example two homology classes in some $H_n(X_i)$ map to the same homology class $H_n(X_{i+1})$ due to the introduction of a new boundary. This contradicts the fact that inclusion maps are injective. The induces homomorphisms encode the local topological changes in the homology of consecutive complexes in the filtration. We will introduce the following terminology to help us interpret this information:

- A homology class is **born** if it is not the image of a class in the previous complex in the filtration under i_*.
- A homology class **dies** if its image under i_* is the zero element or when it is merged with another class (they have the same iamge).
- A homology class **persists** if its image under i_* is not zero.

In order to produce a detailed computation of the persistent homology of a filtration we would have to compute all homology groups of all complexes and then compute all

inclusion maps. Doing so by hand is cumbersome and more importantly far too lengthy. We will avoid doing it in favour of presented diagrams of the evolution of the homology classes and appeal to the reader's geometric and topological intuition to argue their correctness.

Let us look at the following example. Show a pretty picture and explain it

Given the persistence homology of a filtration we can pose the question of how we can rank the classes based on their "significance". We are most interested in the classes that persist for a large number of steps in the filtration. Such classes are exactly the ones we consider significant and are said to have high persistence. Ephemeral classes on the other hand are consider to have very low significance and can be neglected. In practise such classes often correspond to statistical noise or sampling error.

To quantify this precisely we will produce the so called persistence pairs. A persistence pair (t_1, t_2) is a pairing of two timestamps - the birth and death time of a homology class. Every class is associated with a pair such as this where t_1 is the birth time, t_2 is the death time and the class has persisted in all $t_1 \leq t_i \leq t_2$. In the cases of classes that never die such as *this one in that example* we will assume that their death time is ∞. We will call such classes essential and others inessential as in [2].

There is a theorem that states that the persistence digram of a filtration encodes all of the information about the persistent homology groups.

Examples

Finally we will describe an algorithm for computing the persistence pairs. It requires us order all of the simplices in the complex $\sigma_1, \sigma_2, ..., \sigma_n$ according to these rules [3].

- σ_i precedes σ_j when σ_j was introduced later in the filtration than σ_i
- σ_i precedes σ_j when σ_i is a face of σ_j

Not instead of having to compute the homology groups of all complexes in the filtration individually and then computing the induces maps we can perform the whole computation in a single matrix reduction. Let D be an $n \times n$ matrix and such that.

$$D[i,j] = \begin{cases} 1 : \text{if } \sigma_i \text{ is a codimension 1 face of } \sigma_j \\ 0 : \text{otherwise} \end{cases}$$

In other matrix D is a matrix that holds the boundaries of all simplicies in a single matrix. It is called the combined boundary matrix. Now we can perform the following reduction just by column operations.

The proof of this algorithm is outlined in [6].

Algorithm 3 Reduce Combined Boundary Matrix

```
for all j ∈ {1, 2, ..., n} do
    while ∃j' : j' < j and low(j') == low(j) do
        Add column j' to column j.
```

Now let us apply this general theory to a Morse theoretic context. Let M be a triangulation of a smoothly embeded 2-manifold in $\mathbb{R}^3$ and let $f : M \to \mathbb{R}$ be a Morse function. From Morse theory we know that the changes in topology can only happen at finitely many critical points of M. Let $c_1 < c_2 < ... < c_n$ be those critical points. Let us now use the sublevel sets M_{c_i} to make a filtration of M.We obtain the following filtration which we will call ascending

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

From this filtration we can produce the following persistent homology chain

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

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.

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

Let us now restrict M to be compact and contractable. This will ensure that the Reeb Graph of M is a Contour Tree. We would like to tackle the claim made in [9] that the persistent homology pairs are equivalent to branch decomposition pairs. We can immediately see that this claim is either false of ill-defined. The major reason for this is that essential homology classes do not get paired. But in the branch decomposition schemes all critical points are paired.

There is however yet more reason to pursue this. Slightly after the paper of branch decomposition was published, there emerged a way to extend the persistent homology scheme so that all critical points get paired. Using this will allows us to directly compare it to the branch decomposition of a contour tree.

4.3 Extended Persistence

CAN THE SAME CRITICAL POINT BE TWO TWINGS?

We have seen from the definition and computations of persistent homology that not all critical points are paired. Those that give birth the essential persistent homology 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 find a natural and intuitive way to pair the essential homology classes. In terms of filtrations of triangulations of manifolds based on the sub/superlevel sets of a Morse functions this means that all critical points will be paired in some homology.

SHOW EXAMPLE

* REDO FIRST SENTENCE * The way we have paired the remaining critical points in the above example is hopefully both symmetric and consistent with our intuition (developed in the example above). But how do we justify doing so theoretically? Enter extended persistence. The main idea behind extended persistence is to follow the ascending pass 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. To justify this algebraically we would like to make this process a consequence of a new augmented chain that starts with the zero homology group and ends with the zero group. This way we have an assurance that every class that is born will eventually die.

Our initial instinct here might be to just directly apply persistent homology twice. Once on the ascending and the on the descending filtration. The problem that arises is in relating the classes of the two different filtrations. We are able to merge both filtrations into a single long chain, but the induced maps of the two filtrations flow in different directions. Here is an example of what would happen if we attempt this

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

The direction of the arrows is accordance with the how the homomorphisms are induced. To verify this consider that $M_{c_i} \subseteq M_{c_j}$ and $M^{c_j} \supseteq M^{c_j}$ for $i \leq j$. This can be remedied if we find a way to reverse the directions of the arrows in the descending filtration. The issue in finding new maps to induce homomorphisms in the opposite direction is that we will sacrifice the our intuition which captures the evolution of homology classes. This is because other induced homomorphisms will not be natural and as such will be harder to interpret. If we wish to keep using natural inclusion maps we must instead resort to changing the homology groups we use. Let us opt for a descending filtration of relative homology groups like so:

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

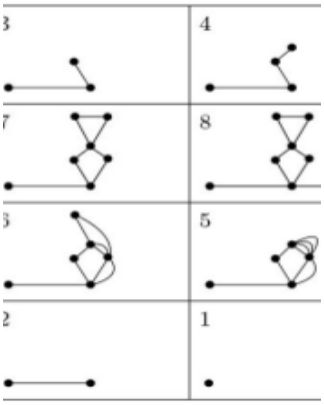
In this relative filtration the homomorphisms are induced by inclusions on the relative homology groups. To see this let (M, M^{c_i}) and $(M, M^{c_{i+1}})$ two consecutive pairs. The inclusion map from M to M takes the superlevel set M^{c_i} in the superlevel set $M^{c_{i+1}}$ because $M^{c_i} \subseteq M^{c_{i+1}}$. By definition [] this is a simplicial map from (M, M^{c_i}) and $(M, M^{c_{i+1}})$ and thus induces a homomorphism between $H_n(M, M^{c_i})$ and $H_n(M, M^{c_{i+1}})$.

The final step to complete our desired sequence is to "glue" these two filtrations together at the point $H_n(M_{c_n}) = H_n(M) = H_n(M, M^{c_n})$. The second equality holds because $M^{c_n} = \emptyset$ and quotienting by the empty set leaves the underlying relative chain complexes unchanged. Putting this all together yields the following chain of homology groups.

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

This augmented filtration justifies the pairing of essential classes according to the intuitive understanding we obtained from example []. The only issue is that the relative homology groups are difficult to interpret on their own. To aid our comprehension of what exactly occurs in the relative filtration we shall employ the Excision Theorem where $H_n(M, M^{c_i}) = H_0(M/M^{c_i}, pt) = \widetilde{H}_0(M/M^{c_i})$ where M^{c_i} is a closed subcomplex of M as required for all $i \in \{1, 2, 3, ..., n\}$.

* Take a look at example how the complex is built and the how it unwinds itself *



4.3.1 Extended Persistence and Branch Decomposition

Now let us examine the relation of extended persistence to branch decomposition. Here is an example branch decomposition of a contour tree of the following 3x3 grid dataset. It is not possible for the branch decomposition to pair the global maximum with the global minimum. There is no monotone path between them. To show that no branch decomposition of a contour tree is equivalent to the extended persistence of the dataset I

will demonstrate that extended persistence necessarily pair the global minimum with the global maximum.

Show the extended persistence filtration of the W3x3 *Show the branch decomposition*

As you can see these methods produce very different pairings indeed. We will conclude that the claims made in [] are either false of not well defined. It so happens that a w-structure that this dissertation is devoted to cause not only computational trouble, but also serve as counterexamples and pose theoretical difficulties as well. To further cement statement we will us show a more general general result in the following chapter which holds for all filtrations of path-connected spaces.

4.3.2 Extended Persistence on Path-Connected Domains

The final step we take on this journey will be to prove a more an original and more general results that will solidify our claim completely.
Proposition 4. *In the extended persistence of a Path-Connected domain the global minimum pairs with the global maximum in the 0th homology*

Proof. Let M be a Path-Connected domain and let $M_1 \subseteq M_2 \subseteq ... \subseteq M_n$ be a filtration of M. This filtration induces etended persistence

$$0 = H_0(M_{c_1}) \to ... \to H_0(M_{c_n}) = H_0(M) = H_0(M, M^{c_n}) \to ... \to H_0(M, M^{c_1}) = 0.$$

As M is Path-Connected it has one path-connected component and therefore $H_0(M) = H_0(M_0) \simeq \mathbb{Z}_2$. Our aim here will be to show that all of the $H_0(M, M^{c_i})$ are trivial. This will mean that the single homology class that exists in $H_0(M)$ will die at $H_0(M, M^{c_n})$ which is exactly the global maximum.

As a corollary of the Excision Theorem we have that

$$H_0(M, M^{c_i}) = H_0(M/M^{c_i}, pt) = \widetilde{H}_0(M/M^{c_i})$$

where $pt = M^{c_i}/M^{c_i}$.

Now let us explore the reduced homology of the topological space M/M^{c_i}. We will show that is it path-connected and therefore the reduced homology is trivial.

By definition M is path connected. Consider the function $\pi : M \to M/M^{c_i}$ that takes a point to it's equivalent class. By point set topology [] we know that π is continuous. We

can also infer that π is surjective. Indeed there there is no equivalence class that no point maps to. Furthermore the continuous image of a path connected is connected by []. As we have that M is path-connected therefore $\pi(M) = M/M^{c_i}$ is path-connected.

By [] we have that $H_0(M/M^{c_i}) = \mathbb{Z}_2$ and by [] that $H_0(M/M^{c_i}) = \widetilde{H}_0(M/M^{c_i}) \bigoplus \mathbb{Z}_2$

We can conclude that $H_0(M, M^{c_i}) = \widetilde{H}_0(M/M^{c_i}) = 0$

Therefore the map induced by the inclusion of the pairs $(M, \emptyset) \to (M, M^{c_n})$ will map the essential homology class of $H_0(M_n)$ to zero. This mean that the global minimum pairs with the global maximum.

□

Following this proposition we can only conclude that in any contractable domain with a w-structure that separates the global minimum with the global maximum branch decomposition is not the same as extended persistence.

4.3.3 Extended Persistence and Join/Split Trees

In this final section we will present examples for how extended persistence on the 0th homology is actually equivalent to computing join and split tree.

Don't know if I can prove this yet, but it probably won't be too hard.

Should I keep this or not?

Chapter 5

Conclusion

This has been a thrillig journey through the depths of algebraic topology. We have ventures through arcane domains where few men and women have ever set foot. We came out victorius and our battle scars will remind us of the glorious adventures of the past.

References

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

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

[3] H. Edelsbrunner and J. Harer. Persistent homology - a survey.

[4] R. Ghrist. Elementary applied topology.

[5] A. Hatcher. Algebraic topology.

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

[7] D. Kozlov. Combinatorial algebraic topology.

[8] D. Parikh, N. Ahmed, and S. Stearns. An adaptive lattice algorithm for recursive filters. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(1):110–111, 1980.

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

Appendices

Appendix A

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 B

Ethical Issues Addressed