School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Something W this way comes

Petar Hristov

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

<Session>

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

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

Type of project: _____

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

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

(Signature of Student) _____

Summary

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

Acknowledgements

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

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

Contents

Chapter 1

Introduction

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

Modern science relies heavily on data collection and analysis. The only way to test a scientific hypothesis in practise is to conduct an experiment that is in accordance to the theory and to collect data outputted by the experiment. Through analysing the collected data scientists can reason about the probability of whether their hypothesis is correct. If that probability is not high enough the hypothesis is rejected and a better one is sought. The analytical current machinery for doing so is reliable and robust. It is based on statistics, machine learning and *other stuff*. One major issue is that more data can be collected than can be analysed []. The only solution to this problem is to reduce the amount of data. The central problem with this approach is in picking which pieces of data to discard. Ideally one would like to discard data which is of little significance and would slow down the analysis with little to offer in terms of important insight and information.

This is where computational topology can come in. Topology is about the qualitative features of spaces. These are the big scale general features that are essential in making a space what it is. Computational Topology is concerned with extracting those from data. This is why it is useful in our case.

An invaluable tool in learning about data is visualisation. In fact the same problem hold with scientific and information visualisation. The human eyesight has limited capabilities. Ensuring only relevant and important information is displayed is crucial in the success of such methods. Let us take for example a tool such as the contour tree. It is primarily used as a discrete summary of the connectivity of a scalar field.

Show example and explain it.

Through the contour tree we can accomplish the following amazing things. Give examples.

The first aim of this dissertation will to be aid in the speeding up of the construction of the contour tree from raw data. In the modern world where individual cores of processors are not expected to become much faster practical speedup has to come from leveraging parallelism be it distributed or shared memory.

The second aim of this dissertation is to discuss the simplification of contour trees and how that can be approached from two different perspectives in the realm of

computational topology.

Computational Topology

What is computational topology? What are the primary tools that are used? Where have they been used in practise and with what success.

Talk about why performance can be lacking. Talk about parallel algorithms. Talk about the future of computation.

Outline what you will do in the thesis. Go though the contents in the chapters.

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.

Chapter 2

Background

The two key concepts we will introduce in this dissertation are the Contour Tree and Persistent Homology. In order to be able to do this we have to first take a step back and walk the reader through a range of other mathematical disciples disciplines. The preliminaries include Set Theory, Point Set Topology, Differential Topology and Algebraic Topology. We will opt for introducing these fields in the context of Computational Topology and give the reader the necessary intuion behind the main definition and results we will use in the following sections.

2.1 Point Set Topology

Contractable Space

Homotopy

2.2 Differential Topology

2.2.1 Morse Theory

Morse Function

2.2.2 Reeb Graph

Reeb Graph

2.3 Algebraic Topology

2.3.1 Building Blocks

2.3.2 Euler Characteristic

Chapter 3

Contour Tree Construction

The Reeb Graph of a contractable topological space is connected and acyclic. This allows us to define this special case of the Reeb Graph as the Contour Tree. In this dissertation we will limit ourselves to practical spacial data analysis where data that is regularly sampled from two or three dimensional Euclidean space. Examples for this are medical imaging [], X-ray crystallography [], terrain mapping data [] and other.

The resolution of any sampling process is limited and if we are to leverage the machinery we have introduced so far we must assume that there is an underlying continuous function present in the space. Our only option in this case is to construct an approximation of this function based on the values we have sampled. This is usually done by constructing a simplicial complex where the data points are the vertices and we add higher dimensional simplicies to completely fill the space between them (see fig[]). The resulting data structure we will call a simplical mesh. The values of the approximation function on the simplicies are obtained via linear interpolation between the verticies of each simplex.

Add more info on simplicial mesh

As long as the original values we have sampled are unique we can show that the resulting linear interpolation function is a Morse function and that the critical values are critical points are the vertices of the mesh. This is one of the key properties that enables efficient computation of the contour tree. We will discuss how to deal with non unique values in the coming sections.

3.1 Algorithms for Computing Contour Trees

In this section we will present the state of the art in current algorithm for contour tree construction. They are based on the work of [].

Explain more algoirthms.

Expand on this!

3.2 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 1. *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 2. *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.

3.3 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 computed. We will describe for the process for the join tree only as it is completely analogous in the split tree case.

Definition 3. *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.

simplicial mesh

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 $v_1 b$ 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 4. *Property of leaves from Hamish's dissertation*

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 *

3.4 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 linear as well.

3.5 Parallel Algorithm

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

In the year 2016 there was a paper [] published that introduced a data parallel implementation of the contour tree algorithm. The algorithm follows the same two phases of first computing join and split trees and them combining them. We will neglect the details in the new data-parallel way of computing join and split trees as they are not as relevant to the dissertation and the problem we are addressing.

The merge phase of the new algorithm has remained large unchanged. The only difference is that leaves are batched and processed in parallel. This is where the bottleneck of the parallel algorithm lies. Imagine a long chain such as in example []. If we do this approach then our parallel algorithm will be no better than a serial one. We are bottlenecked by the structure of the tree.

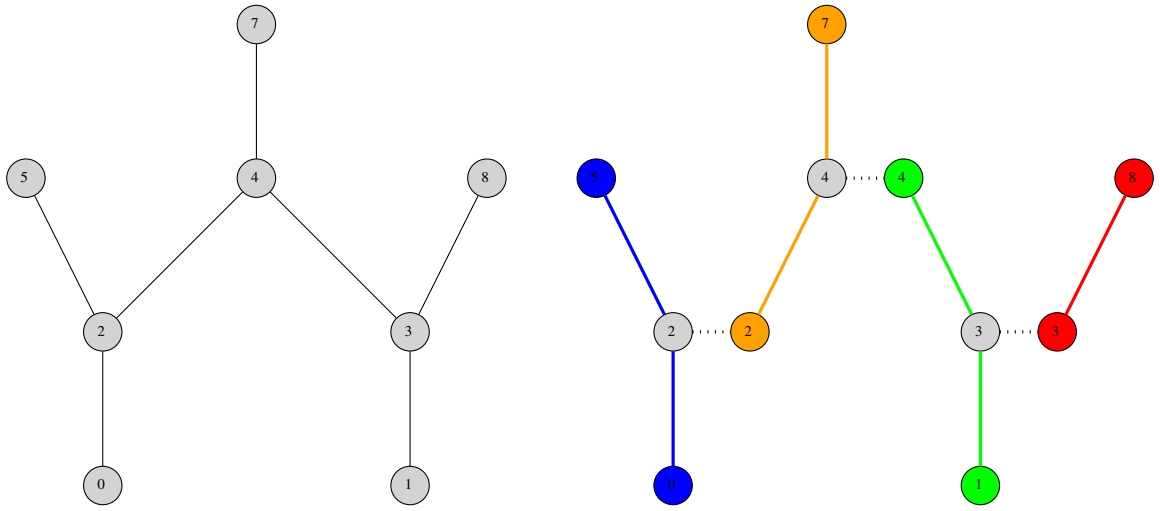
On the other hand if we are able to collapse long chains such as that then effectively we obtain a tree with no vertices of degree two. By out lemma [] we have that at least half of it's vertices will be leaves and all leaves can be processed in parallel.

There is a way to collapse chains with a monotone paths. This is a short description of it. There is however no way to collapse non monotone chain. Therefore long chain with many zig zags in them serialize the parallel algorithm. Out main goal will be to explore long zig zag chains such as that one and try to find a way for the algorithm to collapse them and have logarithmic collapse.

* Explain the zig zags better, but don't go into too much detail until the next chapter.*

3.6 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



(a) Contour Tree. (b) Branch Decomposition.

Figure 3.1: Branch Decomposition of a 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 [12]. 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 pairs 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 [12] cites that the persistence defined in that way is similar to persistence first defined in [7]. In Chapter N of this dissertation we will demonstrate that this claim is either incorrect of misleading. Stay tuned folks.

Isolated, Nice

Chapter 4

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.

4.1 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.

4.2 W-Paths in Height trees

We will now 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.

4.2.1 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 1. *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.

4.2.2 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:

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

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.

4.3 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 5. *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 6. *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.

4.3.1 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 2. *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:

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

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

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

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

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
```

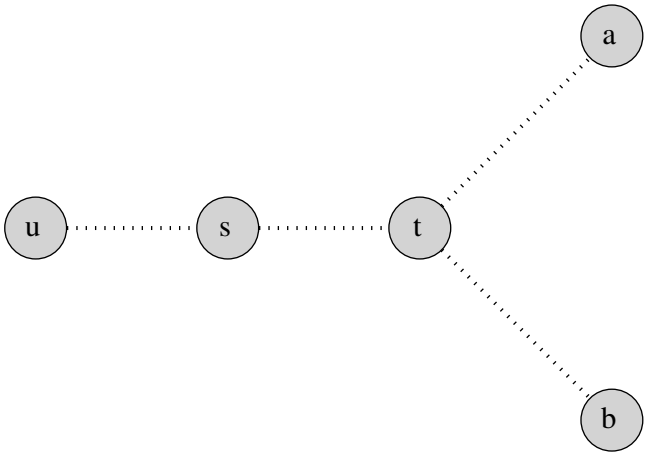


Figure 4.1: Relative position of vertices in Case 1.1

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

The proof can then be split into several cases depending on the relative positions of s, t, a, b and u.

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

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

In this case $s \rightsquigarrow u$ is a subpath of $t \rightsquigarrow u$, which in turn means that $w(t, u) \geq w(s, u)$. By equation 4.2 we also have that $w(s, u) \geq w(s, a)$. We can therefore conclude that $w(t, u) \geq w(a, t)$ as $s \rightsquigarrow a$ is a subpath of $t \rightsquigarrow a$.

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

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

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

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

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

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

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

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

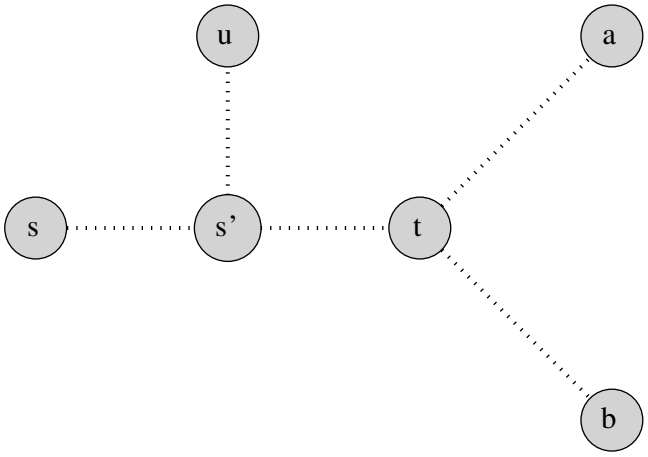


Figure 4.2: Relative position of vertices in Case 1.2

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

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

Considering all possible values that x and y can take, we can see that the minimum value for the right hand side of the inequality is at $y = 0$ and $x = 1$. The final conclusion we may draw is that $w(u, v) \geq w(a, b) - 1$.

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

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

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

where $x, y \in \{0, 1\}$ indicate whether s' is a kink in the corresponding path as before. By equation 4.1 we know that $w(s, u) \geq w(s, a)$ and therefore:

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

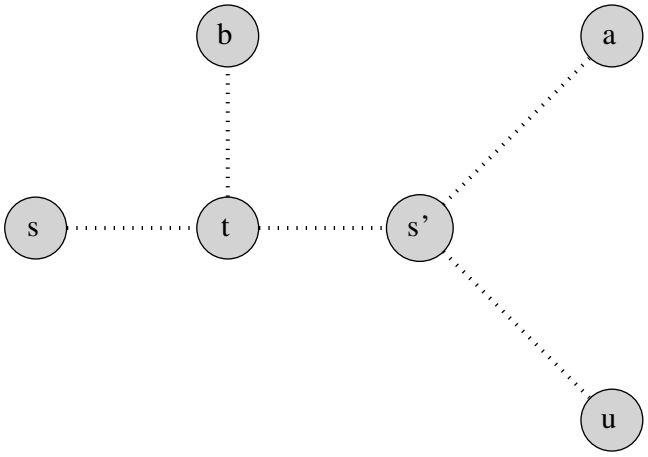


Figure 4.3: Relative position of vertices in Case 2 (t could be equal to s').

Since s' lies on the path from t to u we have that $w(t,u) \geq w(s',u)$ by the subpath property. We can use this to conclude the following:

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

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

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

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

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

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

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

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

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

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

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

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

We can do a path decomposition as follows:

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

As $w(s,u) \geq w(s,a)$ (by equation 4.2)we obtain that:

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

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 3. *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 4. *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. □

4.3.2 Pathological Cases in 2xBFS

Here are examples of the pathological case where the w-diameter outputted by the 2xBFS algorithm is different than the actual w-diameter. Look at Fig. 4.4. There is one example corresponding to all proof cases. In all the examples the vertex outputted by the first BFS function would be u after which the longest path would be outputted as $u \rightsquigarrow a$ or $u \rightsquigarrow b$. The problem is that $w(u, a) = w(u, b) = 1$, but $w(a, b) = 1$ or 2 in the first example. Note that even if we adapt the algorithm so that it finds the vertex with largest w-length away and breaks ties with normal distance, we will still be able to apply these examples. We just need to augment them by subdividing the path to u to make if further away from a and b, but keep the same w-lenght between s, a, bu.

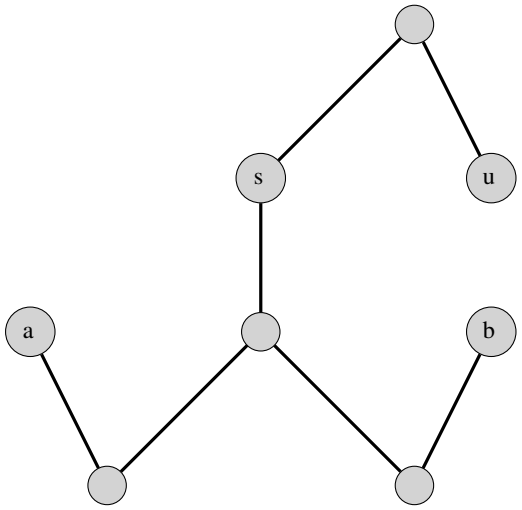
4.3.3 Attempts at resolving the accuracy of 2xBFS

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.

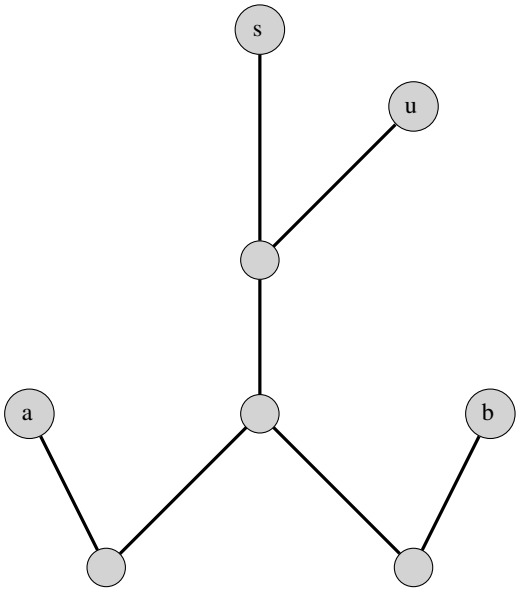
One key observation we can make is that on the second run of the BFS we get a w-path that is necessarily longer or equal to one found in the first BFS search. A natural question to ask is whether running the BFS a third, fourth or for that matter nth time would result in the actual w-diameter. On every 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.

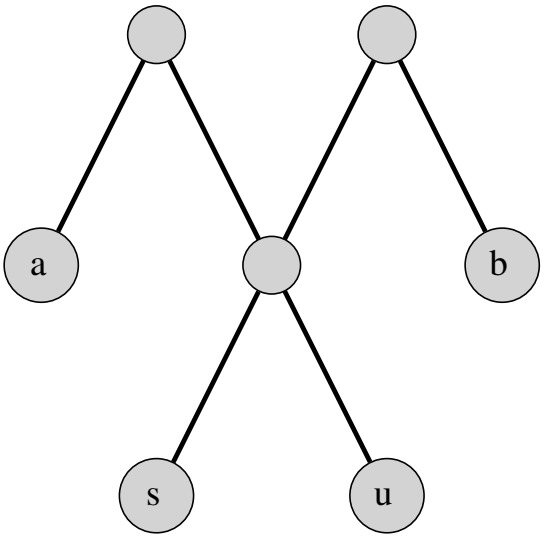
In the search for a better solution let s be a starting vertex and let the vertices
$U = \{u_1, u_2, ..., u_n\}$ be the furthest away in terms of w-distance and
$W = \{w_1, w_2, ..., w_n\}$ be the ones second furthest away. By the proof of the algorithm we



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



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



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

Figure 4.4: Branch Decomposition of a Contour tree.

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!*

4.3.4 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.

Here is the pseudocode for the algorithm.

* Not sure if this is enough *

Let us not prove that the algorithm is correct.

Algorithm 2 Computing the W Diameter of a Height Tree.

```
1: Function W_DFS(T, s)
2: // Base Case
3: if |T.Adj[s]| == 1 AND s.π ≠ s then
4:     s.W = 0
5:     s.w = 0
6:     return
7: // DFS Visit
8: for all u ∈ T.Adj[s] do
9:     if u.π == ∅ then
10:        u.π = s
11:        W_DFS(T, u)
12:
13: // After all neighbours are visited
14: // Calculate w-height of s
15: for all u ∈ T.Adj[s]/s.π do
16:     if L[u] == ∅ then
17:         H[s] = max(H[s], H[u]);
18:     else
19:         for all v ∈ L[u]/u do
20:             H[s] = max(H[s], H[u] + w_{v,s}(u));
21: // Find all children that contribute to the a w-height path
22: for all u ∈ T.Adj[s]/s.π do
23:     if L[u] == ∅ AND H[s] == H[u] then
24:         L[s] = L[s] ∪ u
25:     else
26:         for all v ∈ L[u]/u do
27:             if H[s] = max(H[s], H[u] + w_{v,s}(u)) then
28:                 L[s] = L[s] ∪ u
29: // Find the maximum path combination
30: maxCombine = 0
31: for all u ∈ T.Adj[s]/s.π do
32:     for all v ∈ T.Adj[s]/s.π do
33:         if v == u then
34:             continue
35:         temp = H[u] + H[v]
36:         if L[u] ≠ ∅ then
37:             for all t ∈ L[u]/u do
38:                 if w_{t,s}(u) == 1 then
39:                     temp = temp + 1
40:                     break
41:         if L[v] ≠ ∅ then
42:             for all t ∈ L[v]/u do
43:                 if w_{t,s}(v) == 1 then
44:                     temp = temp + 1
45:                     break
46:         if w_{u,v}(s) == 1 then
47:             temp = temp + 1
48:         maxCombine = max(maxCombine, temp)
```

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

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

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

Proof. Let s be the root of the current subtree T_s in the computation of the algorithm. Suppose that our algorithm has identified that the longest path through s has t kinks. Suppose for the sake of contradiction that there is another w-path with at least $t+1$ kinks that goes through s.

That path must go through two of the children of s. Let those children be u and v. We must have that the value of $h(u) + \max_{t \in L_u}(w_{s,t}(u))$ is maximum amongst all children of s. Otherwise we could pick a bigger one to create a longer w-path with v. But if u is such a vertex then either it or one with the same value would have been picked by our algorithm. This argument also holds for v by symmetry.

We have found that our algorithm either must have computed either u and v or children s equivalent to them. By the way we combine paths we always take the maximum children which if possible form a kink with s. Therefore the maximum w-path that passes through s has exactly t kinks.

Contradiction!

* Alternative I can proof this without contradiction just by showing directly any w-path is smaller than the one produced by my algorihtm*

□

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

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

□

We will now take a look at the time complexity of the proposed solution. We can summarise it in the following formula 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.

[11]

Chapter 5

Homology

5.1 Homology

The guiding principle behind the Euler Characteristic was to decompose a space into cells, count them and perform cancellations based on the parity of the dimension of the cells. This approach yields 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. Homology is a tool that was first developed to measure the topological complexity high dimensional manifolds [7] by identifying what other manifolds can be embedded or surround them. For example with homology we can detect that there is a hole in the torus.

The theory of Homology comes in two flavours - **simplical** and **singular**. Simplicial homology is geared towards analysing simplical complexes and singular homology is the appropriate generalisation for arbitrary topological spaces. In this dissertation we restrict attention on singular homology because we are primarily interested in the computational aspect of homology. For this weed tractable spaces. We will however on occasion refer to singular homology when we desire to leverage a more general result from the relevant theory. More information on singular homology can be found in the following sources [6, 5]

Homology is built around the interplay between the two key concepts of **cycles** and **boundaries**. Let us consider the simplical complex depicted on Figure 5.1 as an example. It consists of four vertices $\{a, b, c, d\}$, five edges $\{ab, bc, ca, db, cb\}$ and one face $\{abc\}$. The boundary of a simplex consists of its codimension-1 faces. For example the boundary of the 1-dim simplex ab consists of the 0-dim simplices a and b. The boundary of the 2-dim simplex abc consists of the 1-dim simplices ab, ac and cb. A cycle on the other hand consists of the simplices that form the boundary of a simplex that is of one dimension higher (regardless of whether that bigger simplex is in the complex). In our example we can observe that the edges ab, bc, ca and bd, dc, cb form a 1-dim cycle. 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

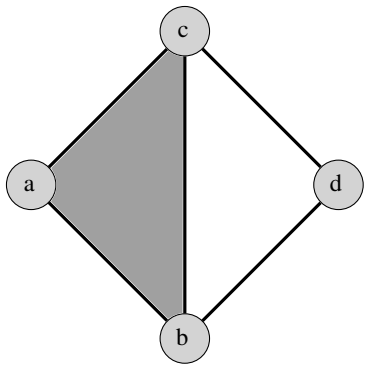


Figure 5.1: An Example Simplical Complex

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. 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.

The interplay between between cycles and boundaries is in asking the question - which cycles in the complex are **not** the boundary of a higher dimensional simplex. This is important because these cycles cannot be contracted to a point. The lack of higher dimensional simplex the enclose means there is a void of some dimension in our simplex.

Let us be more formal now. To begin with, we will operate with vector spaces over the field of coefficients $\mathbb{Z}_2 = \{0, 1\}$ together with the standard operations of addition and multiplication modulo two. We could develop the same theory with the familiar coefficient in $\mathbb{R}$ but we would sacrifice geometric intuition and computational efficiency later on. The building blocks of the homology of a simplical complex X are:

- The **vector spaces 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 for $n \in \{0, 1, 2, ..., \}$.
- The **boundary maps** of the n-chains of X. These are linear maps between consecutive vector spaces of n-chains. It is denoted as $\partial_n : C_n(X) \to C_{n-1}(X)$.

In our previous example $C_0(X)$ was 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 $\sigma \in C_0(X)$ be a vector, then we can express it as $\sigma = \alpha_0 a + \alpha_1 b + \alpha_2 c + \alpha_3 d$ where $\alpha_i \in \{0, 1\}$ for every $i = 0, 1, 2, 3$. Going a dimension up $C_1(X) = span(\{ab, bc, ca, cd, bd\})$. As we pointed

out earlier the cycle that consists of the edges bc, cd, db is represented by the sum or linear combination $bc + dc + bd = 0ab + 1bc + 0ca + 1cd + 1bd$ and has coordinates $(0, 1, 0, 1, 1)$ in $C_1(X)$ with respect to the basis we have chosen.

* Badly explained redo this * We may of course wish to work with to use a different basis for some of the n-chains. The operation of 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. We can use any linear combinations of the simplicies so long as they have the same span and form a basis. 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 $[v_0, v_1, ..., v_n]$ then we define it's boundary as:

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

We would also like to extend ∂ linearly. 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 between. We will call it a chain complex. This is the quiver representation of a chain complex of a simplicat comples of dimension n.

$$C_n(X) \xrightarrow{\partial_n} C_{n-1}(X) \xrightarrow{\partial_{n-1}} ... \xrightarrow{\partial_2} C_1(X) \xrightarrow{\partial_1} C_0(X)$$

For convenience we can extend this chain on both sided with the zero dimensional 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

the

Let us now translate the geometric intuition we have of cycles and boundaries to domain of algebra. The boundaries are provided to us 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 vectors 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 chain have even parity and cancel in our binary algebra. The set of all vectors that go to the zero vector under the boundary map ∂ is precisely 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 respective n-chains. Before learning how the interplay between cycles and boundaries is translated in this setting we must present the following fundamental theorem

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

Proof. We will only sketch the intuitive outline of the proof and refer the reader to [6] for a more complete version.

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

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

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

* Add the definitions of quotient in the appendix *

We can obtain the kernel as a way of detecting the cycles in chain complex. But how can we disregard the cycles that are covered by a boundary? The answer is another algebraic operation. The quotient of the cycles and their subspace the boundaries. What the quotient operation does is to effectively send all cycles that consist entirely of boundaries to zero. *Show example with previous example*. This is precicely how we define the n-th homology of a chain map.

Definition 7. *The n-th homology group of a chain map is vector space* $H_n(X) = ker(\partial_{n+1})/im(\partial_n)$.

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. Connect them to Euler characteristic

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 vectors 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.

* Leave this or not? *

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 [6] 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

Explain Quotient

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

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 [6].

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 1. *Excision - Let X be a topological space, let $A \subseteq X$ and let $U \subseteq A$ where the closure of U is contained in the interiour of A. Then*

A corollary of this is that if A is a close subcomplex of X then $H_n(X, A) \simeq \tilde{H}_n(X/A, A/A)$ 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.

REDEFINE SIMPLICIAL MAPS

5.2 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 8. *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 [6].

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 [9].

Definition 9. *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 [9]!

Definition 10. *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 [9].

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

Definition 11. *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 [6]).

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 $i_* : H_n(X, A) \to H_n(X, B)$.

Chapter 6

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). It is primarily used for measuring topological features such as shapes. This is done by analysing the homology of subsets of the entire space. To accomodate this new concept we must first introduce some additional properties of the relations of homologies of topological spaces and their continous images. 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.

6.1 Persistent Homology

Persistent Homology emerged in the early 2000s in the this work of [7]. 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.

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

A concrete example of this is Figure 6.4.

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

Figure 6.1: Filtration of a Simplical Complex

$i \leq j$ 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 [3].

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 [4].

- 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.

Algorithm 4 Reduce Combined Boundary Matrix

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

The proof of this algorithm is outlined in [7].

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.

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

Finally we will define what we call the persistence of a homology class.
Definition 12. *The persistence of a single homology class in persistent homology that is born at time t_1 and dies at time t_2 is $t_1 - t_2$.*

There are two primary ways to visualize the persistence pairing produces by persistent homology. The first in through persistence diagrams [3]. In the persistence diagrams the pairs are visualised as points in the Carthesian coordinate system above the main diagonal $y = x$. Example []. The second way is through barcode diagrams. You can see the barcode digram on fig[]. For every simplical complex in the filtration we have a number of starting lines equal to the number of generators for the homology classes. These lines feed into each other according to where the induced by inclusion homomorphism take them in the next simplical complex.

Furthermore when two classes merge we must choose which one survives and which one dies. By established convention [] we will use the Elder Rule. By the Elder Rule the class that was born first continues to persistent and the younger one is destroyed.

6.2 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 $H_n(M, M^{c_{i+1}})$.

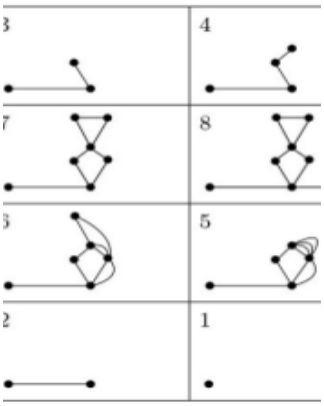
The final step to complete our desired sequence is to "glue" these two filtrations together at the point $H_n(M_{c_n}) = H_n(M) = H_n(M, M^{c_n})$. The second equality holds because $M^{c_n} = \emptyset$ and quotienting by the empty set leaves the underlying relative chain complexes

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 *



6.2.1 Extended Persistence and Branch Decomposition

In this subsection we will tackle a claim that has been made in the paper that first introduced branch decomposition of contour trees. The claims is the following : "... we define the persistence of a branch to be the greater of its length and the persistence of each of it children. This definition differs from the definition of persistence given in [10] because it takes into consideration the topological obstructions. " [12]. In that quote the reference to "[10]" is to the original paper that introduced persistent homology [7].

Let us unpack the claim piece by piece. Firstly the paper only redefines how the persistence of the branches is computed and branches are pair of critical points. Therefore for this to be equivalent to the persistence pairs, both methods must produce equivalent pairs. It is clear from this definition that the actual persistence pairings stay the same, we only compute the persistence value in a different way. What we will aim to show is that the pairings produced by persistent homology are in themselves different from the pairings produces by branch decomposition. After this it will not matter how the actual persistence of branches is computed as they are fundamentally different

things. Lastly the paper claims that the branch decomposition definition differs from persistent homology in that "it takes into consideration the topological obstructions.". It is not clear exactly what the authors meant by "topological obstructions". These obstructions are not defined in the paper nor in subsequent publications. This does not stop us from working with the definition because we will demonstrate a second way in which persistence differs from branch decomposition.

The first major difference we can find comes from the fact that persistent homology as it was originally defined does not pair all critical points. As we discussed previously the pairings of essential homology classes are not defined because they never die in the filtration. In the case of contour trees we will be dealing will simply connected contractable domains. Such domains have a single connected component and therefore have exactly one essential homology class in the 0th homology. This is the class that is born at the global minimum, or the first simlicial complex in the filtration. In branch decomposition on the other hand all critical points are paired.

This is already of major difference between the two. Furthermore the paper on branch decomposition clearly cites the original paper for persistent homology and not the subsequence paper on extended persistence where this issue is remedied. In fact it cannot reference that paper. The branch decomposition paper has been published in January of 2004 and the paper that clearly defines Extended Persistence as such has been published in January of 2009 [8]. There is however more to this story. The concept of pairing critical points unpaired by persistence goes further back than the 2009 paper and can be traced to the paper [10] which was also published in January of 2004. The paper outlines the initial concept of extended persistence in the specific case of 2-manifolds embedded in $\mathbb{R}^3$. This means that the idea of pairing all critical points was present when the claim was made. All of lets us the take this a step further. We will now test whether branch decomposition is equivalent to extended persistence.

We will demonstrate that they are not with a small counter example. This counter example will show that at least one pair in both is necessarily different. In contour trees where the global minimum and the global maximum are not connected via a monotone path branch decomposition cannot by definition pair them as the endpoints of a branch. Extended persistence on the other hand does pair them as the global maximum is the first class in the descending relative filtration that is homologous to the essential class born at the global minimum. To reduce the stress caused by the reader's feverish anticipation of this counter example we will foreshadow that it is a contour tree with a w-structure. This w-structure is what separates the global minimum from the global maximum.

Let us first begin with an example data set show in fig. Let us call that X. The contour tree of this data set is shown in fig and a branch decomposition of this contour tree is

shown in fig. The ascending filtration of this data set is shown in fig. The ascending filtration consists of nine simplical complexes $\{X_1, X_2, ..., X_9\}$. According to that filtration we can produce the persistence diagram on fig[]. The extended persistence of the 0th is shown in fig.

According to our extended persistence computation the pair are $(2, 5)$ and $(1, 9)$. The first pair comes from ordinary persistence. We can see on fig [] that a component is born in time 2 and dies in time t. The second pair is of the global minimum and global maximum. It comes from extended persistence. To verity this observe that $H_0(X_9) = \mathbb{Z}$ because there is one one connected component. The next homology group in the sequence in the group is $H_0(X, X^9) = \widetilde{H}_0(X/X^9) = \widetilde{H}_0(X/\{9\}) = \widetilde{H}_0(X) = 0$. It follows that the induced map $i_* : H_0(X_9) \to H_0(X, X^9)$ is the zero map. Therefore the homology class in $H_0(X, X^9)$ that is born at time 1 at the global minimum must die time 9 and so $(1, 9)$ is a pair in extended persistence.

Here is the summary of you findings.

- There is no monotone path between the global minimum and global maximum in the data set.
- There is no monotone path between the global minimum and global maximum in the contour tree.
- No branch decomposition of the contour tree can pair the global minimum and the global maximum.
- Extended persistence pairs the global minimum and the global maximum.

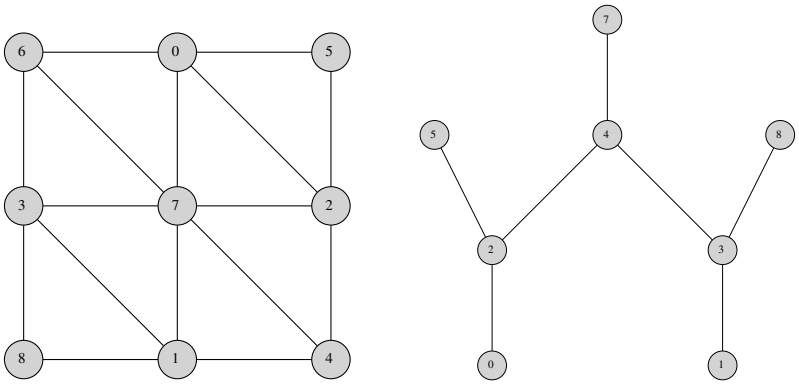
This counter examples demonstrates that branch decomposition is not always equivalent to extended persistence. But they are equivalent on some contour trees. Examine the simple contour tree on fig []. Let us now go a step further and demonstrate a whole class of counter examples where they are not equal. This class will be that of all contour trees where there is no monotone path between the global minimum and the global maximum. It is already clear that in the branch decomposition of such trees they will not be paired. All we have to do is prove that extended persistence does pair them.

Here is the ascending filtration.

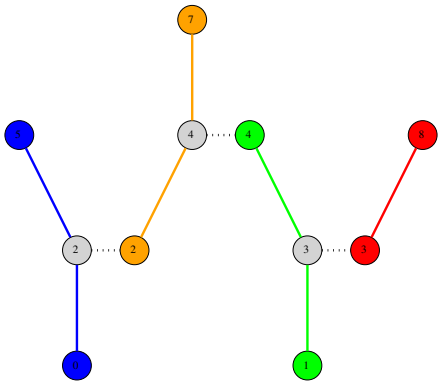
6.2.2 Extended Persistence on Path-Connected Domains

The final step we take on this journey will be to prove the following original and more general result.

Proposition 1. *In the extended persistence of filtration a Morse function with a Path-Connected domain the global minimum pairs with the global maximum in the 0th*



(a) Original Data Set (b) Contour Tree



(c) Branch Decomposition.

Figure 6.2: Branch Decomposition of the Contour tree.

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

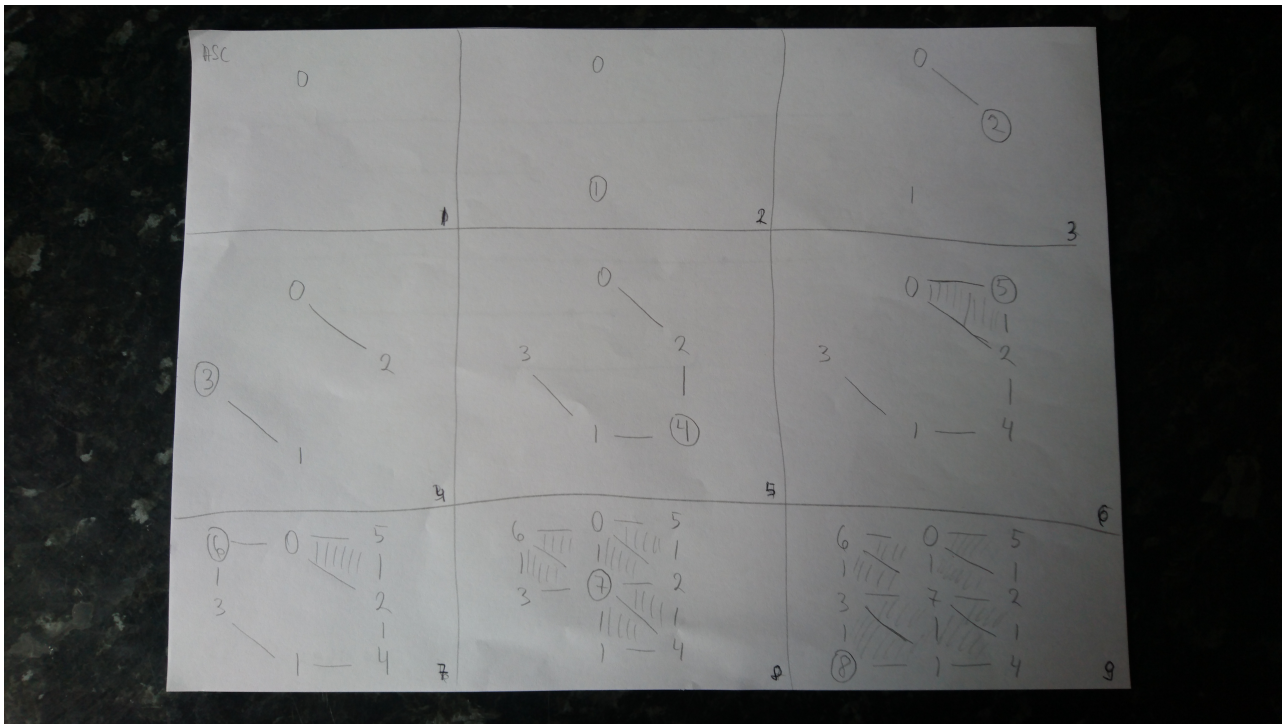


Figure 6.3: Ascending filtration of the Simplical Complex

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.

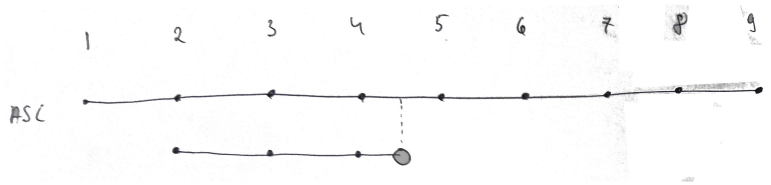


Figure 6.4: Extended Persistence of the Simplical Complex

6.2.3 Extended Persistence and Join/Split Trees - Thoughts on future directions

What we have uncovered is not the complete story. There is a deeper connection between extended persistence and contour trees in general. If we look at alternative algorithms for constructing the 0th homology specifically we can see that the extended persistence of an ascending filtration is equivalent to branch decomposition of the join tree and the extended persistence is equivalent to a branch decomposition of the split tree.

Going back to the claim made [12] we see that there is truth to it if we adjust it to extended persistence and apply specifically to join and split trees. Here future questions that arise based on this fact.

- By using the persistence pairs on other homology group we can produce split and join tree for the cycles, void, etc... We can them combine them like we combine the join and split trees for the contour. What kind of a structure does that yield?
- Can we compute the contour tree directly from homology. Can we find a natural continuous function between the level sets of a filtration. Would the persistence of this sequence be equivalent to the contour tree. In higher dimensions would be equivalent to the previous point?
- Who killed Laura Palmer and will the new Twin Peaks be good?

Chapter 7

Empirical Study

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

In this chapter we will study the w-structures empirically by implementing the algorithms that we can developed in Chapter 2 and using them to analyse real world data sets. The reason for studying them in practise is to learn more about them and explore how exactly they slow down computation. To accomplish this we will set two primary objectives. Firstly to correlate the iterations needed to collapse the contour tree in the merge phase of the algorithm with its w-diameter. This aims to give strength to the hypothesis that the w-structures do slow down computation in practise. Secondly to demonstrate that w-structures are found in real world data and do appear more abundantly as the size of datasets increases.

In addition to out primary objectives there are some additional topics, related to the empirical study, that will be explored

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

7.1 Implemented Algorithms

For the purpose of the empirical study we implemented all three w-diameter algorithms we developed in the previous chapter. Those are the brute force algorithm, the 2xBFS and DP algorithms. In order to produce the contour trees of real life data sets we also implemented the serial version of the contour tree algorithm that is based directly on [2]. To test our implementations for correctness we required the generations of more data sets than we had. This is why we have also written three small utility programs.

One generates randomly populated data grids of size $n \times m$, the second generates random trees on n vertices and lastly one that generates all $n \times m$ permutations of data grids populated with the numbers from 1 to $n \times m$. All algorithms were implemented in C++ and all algorithms are serial.

The first algorithm that we implemented was the algorithm for computing the contour tree. To keep the implementation of this algorithm straightforward we have opted for working with two dimensional data sets only. To ensure the correctness of the implementation we have ran the code against code provided by Dr. Hamish Carr that implements the same algorithm. We have found that the two implementations produce the same output for all data sets we have tested - both from real data and random data.

For the brute force algorithm and the 2xBFS algorithms we based our implementation entirely on the pseudocode provided in the previous chapter. The source code for those can be found in the appendix. For the DP algorithm we had to implemented a bottom up approach because the recursive one we suggested in the previous chapter was not efficient enough for large data sets. To adapt the algorithm to a bottom approach we firstly ran a standard Breadth First Search from a node in the tree. With it we computed the leaves, 1st order leaves, 2nd order leaves and so on. After this we extracted all of the code that was used in the backtracking phase of the Depth First Search and applied it first to all leaves, than 1st order leaves and so on. This ensured that all children of a vertex were computed before the node it self and allowed us to avoid using recursion at the expense of some additional preprocessing and higher memory footprint for storing the order of a vertex.

In order to test the w-diameter algorithms we compared their output to one another because there is no other way to establish the ground truth. We considered the brute force algorithm to be the most reliable because it's correctness if most obviously seen and because it is the easiest to implement and hence reduced the chance of programming error. In all of out tests including real and random data the brute force and DP algorithms produced identical results and the 2xBFS algorithm produces results of no more than two less. This is consistent with the proofs we presented in the previous section.

The utility program that generates randomly populated $n \times m$ grids was straightforward forward and only required two nested loops and the generation of uniform random numbers. The program for generating permutations was done by generating all permutation on the numbers $\{1, 2, ..., n \times m\}$ and them reshaping the permutations into a $n \times m$ array. The program for generating random trees was developed using the union-find data structure. We start off will a graph on n vertices and 0 edges. We start adding edge between randomly selected vertices and we use the union-find data structure to make sure they are not in the same connected component. For if they are

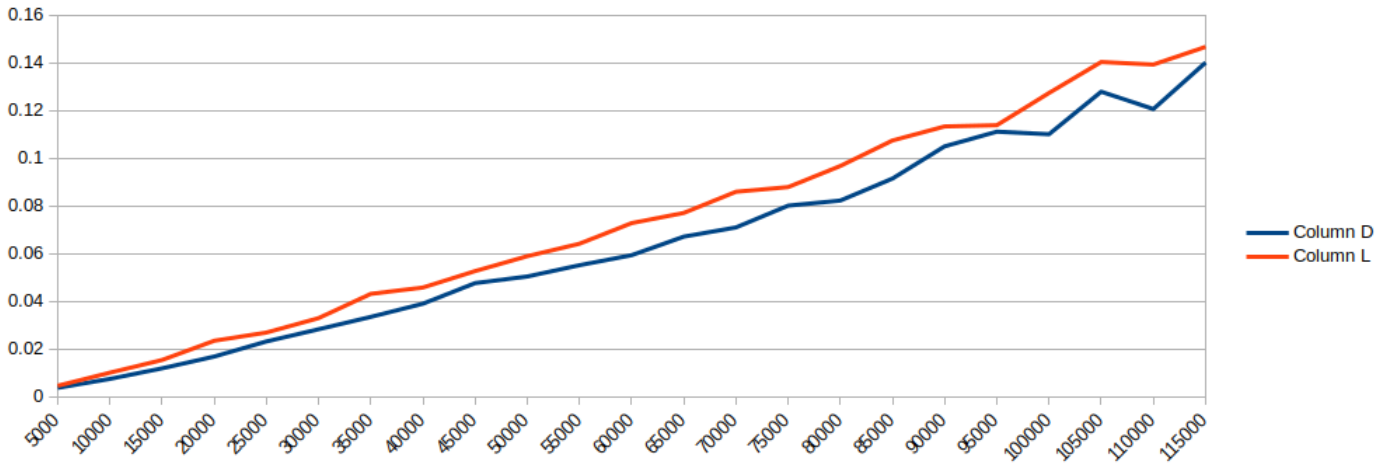


Figure 7.1: Running time of 2xBFS (blue) and DP (red) on randomly generated trees.

we would be introducing a cycle. This process repeats until all vertices are not in the same connected component.

The only third party program that was used was provided by Hamish Carr and it was his implementation of this contour tree algorithm.

7.1.1 Datasets

* Ask Hamish about these. *

7.1.2 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 we have shown to be it's worse case running time. A more detailed algorithmic analysis is needed to prove this theoretically. We will defer doing so for now.

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 on randomly generated data sets is linear.

This is further supported by the statistical analysis through linear regression. *Do line linear regression and output findings*

Let us now analyse the running time of the algorithms on real data. We can see on the following table that the linear relationship between the two still holds. The "Factor" column indicated how many times slower DP is compared to 2xBFS.

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

The running time of 2xBFS is as we can expect linear. The more interesting finding we obtained is that on average in practise the running time of DP is linear as well. Our hypothesis that that the expected running time would drop to linear on trees is proven to be correct by the data. Further tests would include ...

7.2 Analysing w-diameter

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.

7.2.1 Mountain Range Data

This is elevation data taken from the Canadian Mountains. *Ask Hamish about info on the data* *Ask which algorihm to use for iterations*

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

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?

7.2.2 Images

Find some images and test them and write about them.

7.2.3 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.

7.2.4 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

7.3 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.

7.4 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.

7.5 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.

References

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

[2] H. Carr. Efficient generation of contour trees in three dimensions.

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

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

[5] R. Ghrist. Elementary applied topology.

[6] A. Hatcher. Algebraic topology.

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

[8] J. H. Herbert Edelsbrunner and D. Cohen-Steiner. Extending persistence using poincare and lefschetz duality.

[9] D. Kozlov. Combinatorial algebraic topology.

[10] J. H. Y. W. Pankaj K. Agarwal, Herbert Edelsbrunner. Extreme elevation on a 2-manifold.

[11] 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.

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

Appendices

Appendix A

Additional Proofs

Lemma 8. *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 9. *There are at least k vertices for every vertex of degree k in a tree.*

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

□

Appendix B

Vector Spaces, Quiver Diagrams and Barcode Diagrams

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

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

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

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

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

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

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

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

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

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

Show a barcode diagram.

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

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

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

Example 1. *A Simple Chain Complex*

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

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

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

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

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

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

Show a picture of the cosets.

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

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

Appendix C

External Material

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

Appendix D

Ethical Issues Addressed

Appendix E

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

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

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

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

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

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

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

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

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