Wasserstein distance for persistence diagrams

Jacobus Leander Conradi
Vincent Reinthal

November 20, 2019

Lab Report

supervisor: Prof. Dr. Rolf Klein

secondary supervisor: Dr. Elmar Langetepe

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

Table of contents

1 The problem **1**
- 1.1 Persistent homology . . . 1
  - 1.1.1 Persistence Diagrams . . . 2
- 1.2 Wasserstein distance . . . 3
- 1.3 Feature points . . . 4

2 Persistent homology **4**
- 2.1 Computation of the Čech complex . . . 4
- 2.2 Calculation of persistent homology . . . 6
  - 2.2.1 Optimizations . . . 8
  - 2.2.2 Further optimizations . . . 12

3 Modified Wasserstein distance **13**
- 3.1 Efficient Calculation . . . 14

4 Feature Proposals **16**
- 4.1 Feature Detection . . . 17
- 4.2 Feature Selection . . . 17

Conclusion **18**

Notes on the Java Program **19**

Literatur **21**

1 The problem

In this work we want to take a look at the comparison of different types of image data. For this we consider the concept of persistent homology. Persistent homology can best be described as the "recognition of structure while squinting". In essence, we want to reduce a complex-looking data set to basic geometric forms. In particular, this approach is very intuitive and close to the subconscious recognition of structures that are characteristic of humans.

1.1 Persistent homology

A tool well suited for the discrete analysis of rough shapes is homology. This is the case, since homologous or "warped" data produce the same or very similar homology. And this is precisely what we are interested in. In order to calculate homologies, we first need a manifold, or in the case of a computer a necessarily discrete representation, a CW complex. We begin by defining the CW complex we will be working with, namely the Čech complex for a given point set. The Čech complex is interesting for us, since it models the "squinting" part. It can be thought of as letting balls grow around each point from some point set and looking at the resulting shape.

Definition 1 (Čech complex)**.** *Let $X \subset \mathbb{R}^d$ be a finite set and $\varepsilon > 0$ arbitrary but fixed. We construct the Čech complex $\check{C}_\varepsilon(X)$ as follows. The 0-skeleton is simply X itself. Any subset $\sigma \subset X$ is in the $(|\sigma| - 1)$-skeleton iff the intersection $\bigcap_{x\in\sigma} B_\varepsilon(x)$ of balls with radius ε around every point of σ is non-empty.*

By the Nerve-Theorem the Čech complex $C_\varepsilon(X)$ is homotopy equivalent to the union of balls $\bigcup_{x_i\in X} B_\varepsilon(x_i)$ (see [Ghr14]). See fig. 1 for an example of a Čech complex.
Note, that for any finite X there is a $\mathcal{E} > 0$, such that for any $\varepsilon \geq \mathcal{E}$ we have $\check{C}_\varepsilon(X) = \check{C}_\mathcal{E}(X)$. Call this stabilizing CW complex $C(X) := \check{C}_\mathcal{E}(X)$.
We define a filtration f on $C(X)$ via $f : C(X) \to \mathbb{R}$, $\sigma \mapsto \min\{\varepsilon|\sigma \in \check{C}_\varepsilon(X)\}$. Note that this filtration is well-defined, since for every sub-cell $\sigma' \subset \sigma$ it holds, that $f(\sigma') \leq f(\sigma)$. Since X is finite, and hence $C(X)$, $\mathrm{im}(f)$ must also be finite. Let $\mathrm{im}(f) = \{\varepsilon_i|0 \leq i \leq N\}$. We now get a sequence of CW complexes

$$\check{C}_{\varepsilon_1}(X) \hookrightarrow \check{C}_{\varepsilon_2}(X) \hookrightarrow \ldots \hookrightarrow \check{C}_{\varepsilon_{N-1}}(X) \hookrightarrow \check{C}_{\varepsilon_N}(X).$$

To extract information from this sequence, we use persistent homology. Persistent homology is looking at the evolution of the generators of the homology groups, as they get passed from one complex in the sequence to the next.

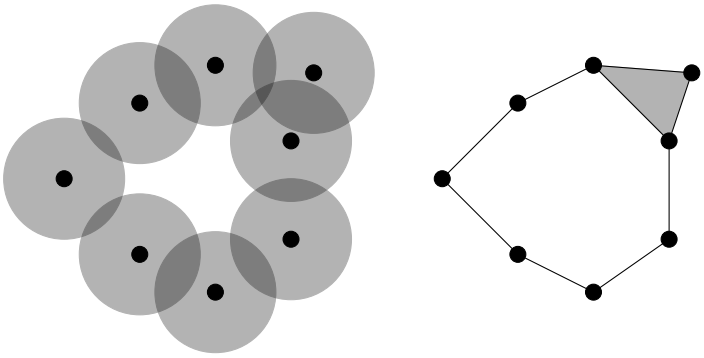

Fig. 1: *A Čech complex for a given ε and generating point set.*

Definition 2 (Persistent homology)**.** *Let*

$$X_1 \overset{\iota_1}{\hookrightarrow} X_2 \overset{\iota_2}{\hookrightarrow} \ldots \overset{\iota_{n-2}}{\hookrightarrow} X_{n-1} \overset{\iota_{n-1}}{\hookrightarrow} X_n$$

be a sequence of CW complex embeddings. For every $k \geq 0$ this implies a sequence of homology groups

$$H_k(X_1) \overset{\iota_1^*}{\to} H_k(X_2) \overset{\iota_2^*}{\to} \ldots \overset{\iota_{n-2}^*}{\to} H_k(X_{n-1}) \overset{\iota_{n-1}^*}{\to} H_k(X_n).$$

For every $0 \leq i < j \leq n$ we begin with $\mathcal{L}_i^j(X)_k$ the set of generators $\sigma \in coker(\iota_{i-1}^*)$, such that $\iota_{j-2}^* \circ \cdots \circ \iota_i^*(\sigma) \neq 0$ but $\iota_{j-1}^* \circ \cdots \circ \iota_i^*(\sigma) = 0$. Now to get the persistent homology from this, we need to apply the so called "elder rule". It may happen, that $\sigma_1 \in \mathcal{L}_i^l(X)_k$ and $\sigma_2 \in \mathcal{L}_j^l(X)_k$ exist (w.l.o.g. $i < j$), such that σ_1 equals σ_2 at some point $t < l$. In this case, we kill the "younger" one of the two, i.e. moving σ_2 to $\mathcal{L}_j^t(X)_k$, or removing completely, if $j = t$. We will call this reduced family $\{\mathcal{L}_i^j(X)_k | 0 \leq i < j \leq n, k \geq 0\}$ the persistent homology of $\{X_i\}$.

In essence persistent homology stores the lives and deaths of various generators. If we input the sequence we got from the Čech complex into the persistent homology, we can analyse the resulting "shape" of the input data after "squinting".
We want to calculate this persistent homology for the Čech complex of given sets of 2-dimensional points efficiently and will take a look at different approaches to calculate these.

1.1.1 Persistence Diagrams

In order to work with persistent homology more intuitively we want to visualize these. A particular type of visualization we want to work with are persistence diagrams. Here we mark the point (t_1, t_2) in $\mathbb{R}^2$ for every generator who is born at time t_1 and dies at time t_2. In more general terms, for every $\sigma \in \mathcal{L}_i^j(X)_k$ mark $(\varepsilon_i, \varepsilon_j)$ in $\mathbb{R}^2$.

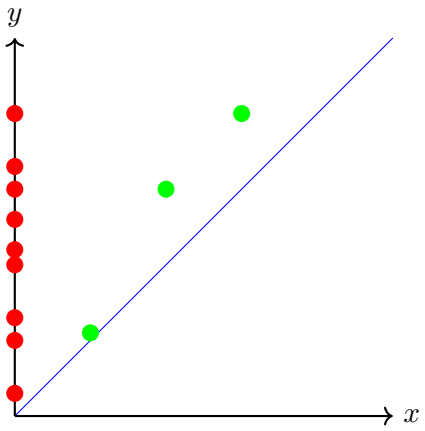

Fig. 2: *A persistence-diagram. Red dots represent the* 0*th, green dots the* 1*st homology*

Note that the points marked in this way are always above the diagonal $\Delta = \{(x, x) \in \mathbb{R}^2 | x \in \mathbb{R}\}$, where points directly on the diagonal represent generators that die quickly after birth, and points far from the diagonal represent generators that live very long. Fig. 2 is an example for a persistence diagram which shows generators of the zeroth and first homology. One can see that the generators of the zeroth homology, which represent connected components, are all born at time 0 and die when the connected components merge.

1.2 Wasserstein distance

Now let two such persistence diagrams be given. To define a distance of these two diagrams, we consider the Wasserstein distance often used in probability theory. This is defined as the "minimum effort to move one probability distribution to another". In our concrete application we define this as follows.

Definition 3 (Wasserstein distance)**.** *Let two finite point sets X and Y with $|X| = |Y|$ be given. Then the Wasserstein distance is given by*

$$W_p(X, Y) = \min_{\varphi: X \to Y} \left(\sum_{x \in X} ||x - \varphi(x)||^p \right)^{\frac{1}{p}},$$

where the minimum is taken over all bijections $\varphi : X \to Y$, with $|| \cdot ||$ being the ℓ_2-norm in $\mathbb{R}^2$.

A special case $W_\infty(\cdot, \cdot)$ we will call the Bottleneck distance.
Notice the restraint $|X| = |Y|$. We want to adapt this such that differently sized sets can be compared with each other as well. For this we want to compare different approaches qualitatively.

1.3 Feature points

As a final step, we will examine methods to find a representative set of points X for a given image to use as input for calculating persistent homology. Again, we want to compare different methods and compare them in the context of persistent homology.

2 Persistent homology

In this chapter we want to focus on the implementation for the computation of the Čech complex, as well as a naïve approach to calculate persistent homology and persistence diagrams. Since we limit ourselves to 2-dimensional point sets X, it is sufficient to limit $C(X)$ to its 2-skeleton, since all higher homologies vanish.

2.1 Computation of the Čech complex

To calculate the Čech complex we first look at the Voronoi diagram for a given point set X in 2-dimensional space. Here $\mathbb{R}^2$ is divided into regions, so that in each region there is exactly one point $x \in X$, and for every other point y in this region x is the nearest point of X to y.
From the Voronoi diagram we can extract the Čech complex. We begin by setting the 0-skeleton to X itself, and the filtration value of every vertex to 0. We add a 1-cell $\{x, y\}$ to the complex, whenever the Voronoi regions of x and y intersect in a line segment. The distance of x and y also gives us the filtration value $f(\{x, y\}) = \frac{||x-y||}{2}$ of the edge. After adding all these 1-cells, we get the planar dual-graph, the DeLauney triangulation of the points from X, when we interpret 1-cells as edges of a graph.
It gets more complicated with 2-cells. Here we have to make an important case distinction. If more than two Voronoi regions intersect, then their respective vertices form a circuit in the DeLauney triangulation and a 2-cell in the Čech complex. Note that in 2 dimensions this intersections is always exactly one point. But what exactly is the filtration value?
This is where the following case distinction comes into play. Looking at the convex hull of the points $x_1, \dots x_{k-1} \in X$, whose Voronoi regions intersect in v, it can happen that v is inside or outside the convex hull. If the point v is inside the convex hull, the filtration value must be chosen as the distance of v to some x_i, which is the same for all different x_j, since otherwise v is not the intersection of these regions. Note that this value $f(\{x_1, \dots, x_{k-1}\}) = ||v - x_1||$ corresponds to the exact value, at which point the "hole" enclosed by balls of radius $||v - x_1||$ around $x_1, \dots, x_{k-1}$ dies.
In the other case, the $k-1$ balls around $x_1, \dots, x_{k-1}$ intersect as soon, as all 1-cells $\{x_i, x_{i+1}\}$ exist, or geometrically all pairs of balls around x_i and x_{i+1} intersect. So $f(\{x_1, \dots, x_{k-1}\}) = \max_{1 \leq i < k} f(\{x_i, x_{i+1}\})$, where

```
public class Voronoi {
    // Stores voronoi vertices, where more than 2 regions
        touch
    private PointD[] vertices = null;
    private VEdge[] edges = null;
    ...
    private void compute(int width, int height) {
         VoronoiResults results =
             org.kynosarges.tektosyne.geometry.Voronoi.findAll(
         sites, new RectD(0, 0, width, height));
         vertices = results.voronoiVertices;
         // Transform output of library to our own data types
    ...
    }
    ...
}
public class ActionGenerator {
    public List<Action> generate() {
        // Generate the list of elements added to the cell
            complex sorted by their filtration values
        ...
        voronoi.forEachVertex(this::computeVertex);
        voronoi.forEachEdge(this::computeEdge);
        actions.sort(Action::compareTo);
        return actions;
    }
    private void computeVertex(PointD vertex, int index) {
        // Create list of actions given the Voronoi Diagram
        ...
        VEdge[] edges = voronoi.getEdges(edgeIndices);
        PointD[] sites = getSites(edges);
        if (Util.isInside(vertex, sites)) {
            actions.add(new FaceAction(...));
            return;
        }
        ...
        actions.add(new EdgeFaceAction(...));
    }
}
```

Fig. 3: *Codesnippet of the generation of* $C(X)$

$x_k = x_1$ for convenience. We want to call these 2-cells *degenerate*. For the implementation in Java we decided to use a library which calculates the Voronoi diagram. We calculate the planar dual graph based on the output of the Voronoi Library. We see in fig. 3, that in the case of degenerate 2-cells we use a `EdgeFaceAction` to insert the 1-cell with highest filtration value of the circuit and the 2-cell in the same time step corresponding to equal filtration values. What exactly insertion means will be covered in Section 2.2.
Note that calling 0-cells vertices, 1-cells edges and 2-cells faces makes sense geometrically.

2.2 Calculation of persistent homology

Since the 1-skeleton is the same as a digraph, we save the complex as a graph and keep an additional list of lists of edges given by the boundaries of 2-cells. And since the second (and higher) homology is always zero, due to our choice of 2-dimensional data, this suffices.
We now insert all the 0-, 1- and 2-cells one by one into the graph/list, according to their filtration values starting with the smallest. These insertions we may also call actions, since later on, we will not only insert but also modify data. This corresponds to growing the balls around each point and looking at the resulting union of these balls. However only the ball-radii or time steps corresponding to filtration values are interesting, in the sense, that something in the structure of the union changes.
To respect the "elder rule", we need to make a choice for 0-cells which vertices are older than other vertices. Since the choice has no influence on the homology or on the persistence diagram, we number the vertices arbitrarily. We use this numbering when calculating the first homology as an orientation of the edges.
The main task in our implementation is the calculation of the first homology. For the zeroth homology we only have to check whenever we add an edge whether the two end points are in the same or different connected components. To maintain the elder rule for these, we keep a reference to the oldest vertex in each connected component. If the endpoints of an added edge live in two different connected components, we update the reference to the oldest vertex in the connected components and remember that a generator of the zeroth homology has died.
Generators of the first homology of $C_\varepsilon(X)$ live in the kernel of the boundary operator $\delta_1 : C_1(C_\varepsilon(X)) \to C_0(C_\varepsilon(X))$, where $C_1(\cdot)$ and $C_0(\cdot)$ are the 1- and 0-cycles of the input complex. The kernel of δ_1 are just all circuits on the graph. Hence we must find all possible circuits in the graph, and then find out which circuits die via linear combinations of these, to compute the first homology. To find circuits, we check whenever we add an edge $e = (v, w)$ (corresponding to the 1-cell $\{v, w\}$) whether v and w are in the same connected component. If this is the case, we find a v, w-path P in $G \setminus \{e\}$. Then

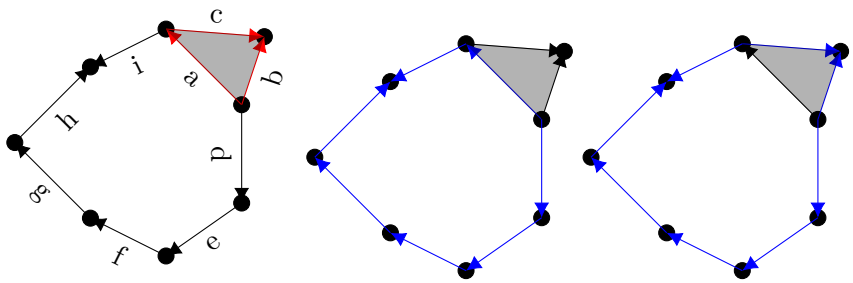

Fig. 4: An example for generators, that die due to a relation. In red is the boundary of a 2-cell, in blue are two generators that are identified with each other via the 2-cell.

we add the circuit $P \cup \{e\}$ to the set of generators of the first homology. At the time this generator is added, e is completely new to the graph and is not yet contained in any boundary of a 2-cell or other circuit, so the circuit $P \cup \{e\}$ is linearly independent from all other circuits and relations until at least the next insertion.
If a 2-cell is added, the circuit describing the boundary of the 2-cell is added to the list. Every time a 2-cell is added, we have to check if a circuit dies. For this we check the kernel of the matrix:

$$\begin{Bmatrix} c_{11} & c_{21} & \dots & c_{n1} & r_{11} & \dots & r_{k1} \\ c_{12} & c_{22} & \dots & c_{n2} & r_{12} & \dots & r_{k2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ c_{1m} & c_{2m} & \dots & c_{nm} & r_{1m} & \dots & r_{km} \end{Bmatrix}$$

Where $c_{ij} = \sum_{e_j \in P_i} 1 - \sum_{e_j * \in P_i} 1$ and $r_{ij} = \sum_{e_j \in R_i} 1 - \sum_{e_j^* \in R_i} 1$, where P_i are the generator circuits, R_i are the circuits describing boundaries of faces, e_j is an edge and e_j^* is the inverted edge.
An example is given in figure 4. We are given two circuits in blue, and a relation in red. If we number the edges as in the first picture and select all circuit orientations counter clockwise we get the following matrix:

$$A = \begin{Bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & -1 & -1 \\ -1 & -1 & 0 \\ -1 & -1 & 0 \\ -1 & -1 & 0 \\ -1 & -1 & 0 \\ -1 & -1 & 0 \\ 1 & 1 & 0 \end{Bmatrix}$$

And thus get a non-empty kernel, $A(1,-1,1)^T = 0$. This corresponds to the two 1-boundaries $P_1 = a - c - d + \ldots + i$ and $P_2 = b - c - d + \ldots + i$

being identified via the boundary of the 2-cell $\delta_2 R_1 = -a + b - c$. This again corresponds to $P_1 - P_2 = -R_1 \in \mathrm{im}(\delta_2 : C_2(C_\varepsilon(C)) \to C_1(C_\varepsilon(X)))$, meaning $P_1 = P_2$ in $H_1(C_\varepsilon(X))$. So we know that the two generators are identified with each other and we may remove the younger one out of the generator list.
We now know when to add edges, faces, how to find all possible generators and how to find "dead" generators. Putting this together we are now able to calculate the first persistent homology.

2.2.1 Optimizations

Since this method is not that efficient ($\sim$ 5 seconds for 100 nodes, $\sim$ 20 seconds for 200 nodes) we optimised it using contractions on the graph and get almost linear runtime ($\sim$ 10 seconds for 1000 nodes). The main idea here is to handle the effect of each relation as operation on the underlying graph instead of maintaining a list of relations. That removes the necessity of computing kernels of a growing matrix in each face step, but is not trivial to implement due to special cases in face contractions.
Let us recap first. We have generated a list of actions, that are of exactly three different types:

- Edge actions: These add exactly one edge between two nodes.
- Face actions: These add a relation going along the boundary of a face, representing the face being contracted.
- Edge-face actions: These exist for degenerate faces and first add an edge of a face before executing the face action.

Since we are going to add and remove edges, we first have to make sure, that we are sorting the actions in a way, that edges are first added and then removed. Logically this should be an implication by the action generation using the Voronoi diagram, but in reality there are sometimes numerical problems, that destroy this order. To do this, we compute a dependency tree, where a face or edge-face action is depending on each edge or edge-face action, that adds an edge of the relation of the face action. Afterwards we sort the actions using this tree order.
Note that since we are now going to contract and remove edges and nodes, we have to allow multi-edges and loops inside the graph. Now we can start to process each action one by one.
Edge actions are the same as before: We add the edge to the graph, independently from whether it is going to be a loop or normal edge. We also have to search for a new circuit in the graph and possibly have to add a new cycle to the set of generators. Due to the way, we are searching for new circuits, we don't even have to change the algorithm when allowing newly added loops.

Edge-face actions at first also add an edge the same way edge actions do, but we omit searching for circuits, since the newly created cycle would die the same time it is born by the face we are about to add. Then we add the face relation just like face actions do.
Before we look at face actions, we have to emphasize the difference between circuits and cycles. Circuits are paths in the graph, where start and endpoint coincide. Cycles are represented by circuits on the graph, but are objects/generators in homology groups.
This leaves us with the evaluation of face actions. We are now starting to modify edges and nodes inside the graph, which means that all future actions and all existing cycles have to be updated in the same way as well. Otherwise the modified graph would not be consistent with existing cycles and upcoming actions. To simplify this, we introduce circuits on the graph, which are defined by a list of oriented edges. Boundaries of faces and cycles can both be represented by circuits, so that we can apply the following operations simply to the graph and circuits. Homology is commutative but since we are representing boundaries and faces as oriented paths, we cannot use the commutativity and have to be more careful about modifying these. The lack of commutativity will later be handled by eliminating linear combinations of cycles.
Let us now discuss how we replace nodes and replace and remove edges. Replacing nodes can for example be done by maintaining a node mapping table, but we decided to do that by having each graph node be an optional pointer to a target node. If for example we want to replace node 1 with node 0 using the elder rule, we have to set node 1 to point to node 0.
Now we have a look at the more complicated replacement and removal of edges. Since we are replacing edges no matter the orientation, we have to replace all occurrences of that edge with the desired orientation and possibly inverse order (see fig. 5 `replace()`). Analogously we remove all occurrences of a single edge no matter the orientation when removing an edge from a circuit (see fig. 5 `remove()`). After both of those operations we try to reduce the circuit as far as possible by successively removing pairs of edges and their inverses (see fig. 5 `revalidate()`).
Now we have all the tools we need to actually evaluate face actions. For the following 4 cases please refer to fig. 6. Furthermore note, that we will denote paths consisting of edges a,b and c in that order by $[a, b, c]$.
Case 1 actually never occurs, since that would mean that the relation is a linear combination of other relations.
Case 2.1 removes exactly one loop and *Case 2.2* occurs at the very start of the algorithm, where nothing has yet been contracted (see fig. 7 top left, bottom left).
Case 3 first searches for a loop, that only occurs once (which is always possible, since otherwise all edges would have been gone through twice, which is impossible in a planar graph). Then it solves the relation for that loop, so

```
public void replace(int edge, int edges) {
    if (edges.length == 0) {
        remove(edges);
        return;
    }
    boolean modified = false;
    // Replace positive and negative orientations of the edge
    int inverse = Graph.inverse(edge);
    List<Integer> positive = Util.toList(edges);
    List<Integer> negative =
        Util.toList(Graph.inverse(edges));
    for (int i = 0; i < size(); i++) {
        int replace = get(i);
        // Replace all occurrences with either positive or
            negative replacement
        if (replace == edge || replace == inverse) {
            this.edges.remove(i);
            this.edges.addAll(i, replace == edge ? positive :
                negative);
            modified = true;
            i += edges.length - 1;
        }
    }
    if (modified) {
        revalidate();
    }
}

public void remove(int... edges) {
    // Collect all edges in positive and negative orientation
    Set<Integer> set = Util.toSet(edges);
    IntStream.of(edges).map(Graph::inverse).forEach(set::add);
    if (this.edges.removeAll(set)) {
        revalidate();
    }
}

public void revalidate() {
    // Iterate through the circuit and reduce inverse
        oriented consecutive edges
    for (int i = 0; i < size(); i++) {
        int j = (i + 1) % size();
        if (get(i) == Graph.inverse(get(j))) {
            edges.remove(Math.max(i, j));
            edges.remove(Math.min(i, j));
            i = Math.max(0, i - (j < i ? 3 : 2));
        }
    }
}
```

Fig. 5: *Codesnippet of replacement and removal of edges*

```
public void addRelation(Circuit circuit, double radius) {
    if (circuit.isEmpty()) { // Case 1
        // Empty relations have no effect
        return;
    }
    if (circuit.size() == 1 || graph.hasNoLoops(circuit)) {
        // Case 2.1, 2.2
        // If the circuit has size 1, the single edge is a
            loop and can simply be removed
        // If the relation has no loops, every edge can be
            contracted and all involved nodes be replaced
            using the elder rule
        remove(circuit.getEdges());
    }
    else if (graph.hasOnlyLoops(circuit)) { // Case 3
        // Find a single loop and replace it with the other
            loops
        replaceLoop(circuit);
    }
    else { // Case 4: Both loops and non-loops
        // Find a non-loop, replace it with the other edges
            and contract all non-loops
        replaceNonLoop(circuit);
    }
    // Kill cycles, that are now empty
    killEmptyCycles(radius);
    // Kill cycles, that are now linear combinations of
        others using the elder rule
    killObsoleteCycles(radius);
}
```

Fig. 6: *Codesnippet of relation evaluation*

that it can be replaced by the resulting sequence of other loops. For example in fig. 7 top right having the relation $[a, b, b, -c]$ we can find a to be a single edge and replace it with $[c, -b, -b]$. That way we get rid of every occurrence of a using the relation and delete it afterwards.
Case 4 is the last case, where some k loops as well as non-loops occur in the relation. Here we search for a single non-loop and replace it using the relation, just like in Case 3. Then we additionally contract all non-loops of the relation. We can do this, since the original boundary of the face was homotopy equivalent to a $k+1$-bouquet. Gluing a face onto a $k+1$-bouquet results in a k-bouquet, where one of the loops gets identified with all other. So we pick one of the non-loops that forms (together with all other non-loops) part of the loop to be identified (see fig. 8 and 2.2.2). We identify all other non-loops with 0 via homotopy, and contract all the vertices to a

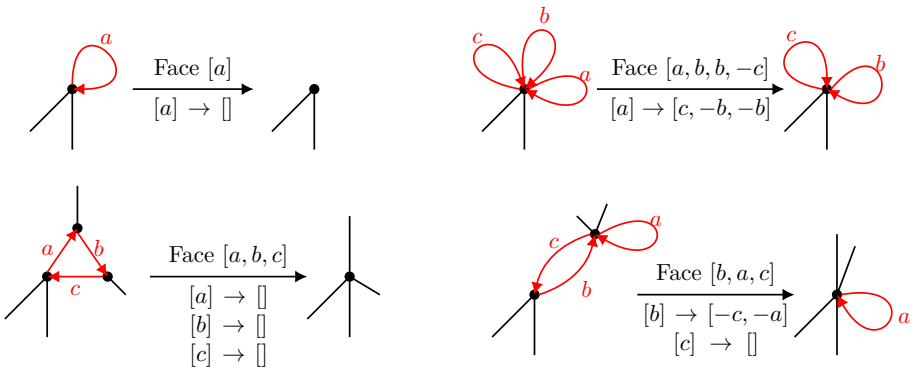

Fig. 7: From top left to bottom right, examples for Case 2.1, 3, 2.2 and 4. The Face is described above the arrow, the resulting additional mappings of edges is described below the arrow.

single vertex. The selected non-loop gets mapped to all the other loops in the bouquet respecting the orientation and then gets deleted as well. In the example in fig. 7 bottom right we find edge b to be occurring only once. We merge the two nodes, delete both edges and add the mappings b to $[-c, -a]$ and c to 0. This way we eliminate all non-loops, that are part of the given relation each time and drastically reduce the size of the graph over time.
Afterwards we kill empty cycles and linear combinations of cycles. The elimination of linear combinations is still done by computing the null space of the edge matrix of all living cycles. However this matrix grows much slower and sometimes even shrinks in size, compared to the other method. This last step is also the part, that respects the commutativity of homology.

2.2.2 Further optimizations

While analysing Case 4 of the previous subsection we discovered, that there is actually more room for improvement. As shown in fig. 8 the actions performed in Case 4 do not destroy any information in the homotopy. The homotopy equivalence (as an example in fig. 8) is given by sending the edges b,c and d to the null-loop. This is a homotopy equivalence, since the contracted set $\{b, c, d\}$ is null-homotopic. And since homology is homotopy invariant (see [Hat02]) this does not change the homology.
However we noticed, that alot of calculations can be skipped, if we respect this homotopy as soon as we add edges. So we changed the algorithm slightly, so that any edge, that does not form a loop gets contracted right after insertion. Only edges that form a loop upon insertion are kept. Then checking for closed paths inside the graph reduces to checking whether the inserted edge is a loop and Cases 2.2 and 4 get obsolete, since no non-loops will ever exist. Also the elimination of linear combinations of cycles becomes

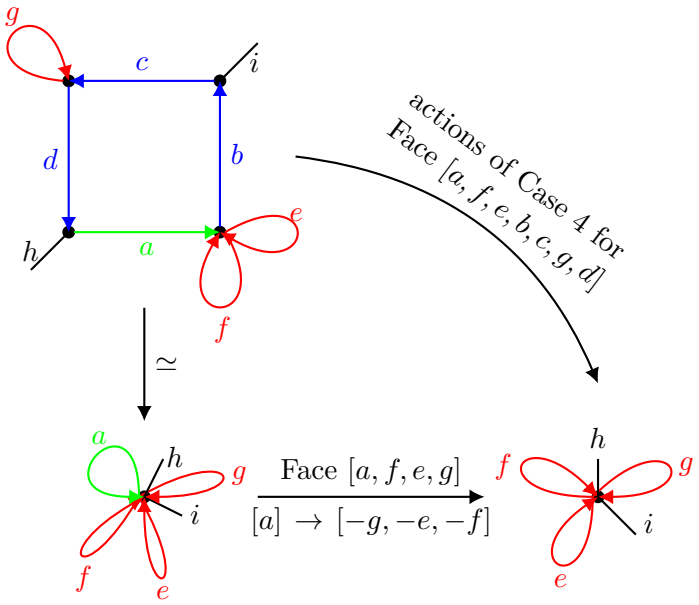

Fig. 8: Correctness of case 4 via a homotopy equivalence.

very easy, since cycles consist of much smaller circuits in the graph. We therefore added an option to compute actions in this manner and observed the computation speed.
It turns out, that this very small modification of only the insertion step regarding edges speeds up the algorithm significantly. While the run time is of course dependent on the instance itself, we experienced a performance increase in this regard of a factor between two and three.

3 Modified Wasserstein distance

First of all we want to extend the Wasserstein distance to differently sized sets. A first idea was to minimize $\varphi : X \to Y$ for $|X| < |Y|$ via injections $\varphi : X \to Y$, and to introduce an error term for every point in Y that is not hit.
Define $\gamma : \mathbb{R}^2 \to \mathbb{R}$, with $(x, y) \mapsto y - x$. The motivation behind this definition is that this is exactly the vertical distance from a point (x, y) to the diagonal. Then we define

$$W'_p(X,Y) := \min_{\varphi: X \to Y} \left(\sum_{x \in X} |||x - \varphi(x)||^p + \sum_{y \in Y \setminus \in(\varphi)} \gamma(y)^p \right)^{\frac{1}{p}}.$$

This approach seemed promising at first, but the error term was too big. The problem is that $||x - \varphi(x)||$ is the ℓ_2 distance of x and $\varphi(x)$, whereas

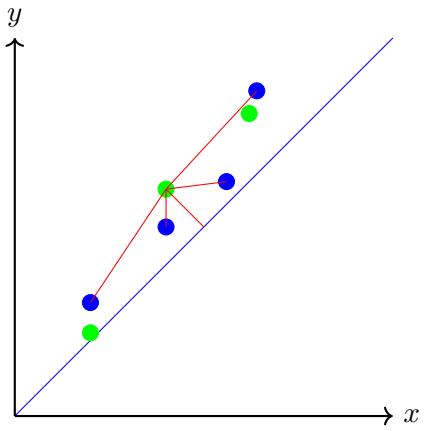

Fig. 9: *The first homology of two different persistence Diagrams in green and blue. Red are all the different distances, a single point of the green diagram has to consider in the mapping problem.*

$\gamma(p)$ is the ℓ_1 distance of p and Δ. Hence we define $\gamma(x,y) = \frac{y-x}{\sqrt{2}}$, to get the ℓ_2 distance of p and Δ instead. However we noticed that this distance is too restrictive. We also want to allow points from X not to be mapped. So now we define the final version of our modified Wasserstein distance, where we apply the error term to any point from X that is not mapped, as well as any point from Y that is not hit.

$$V_p(X,Y) := \min_{Z \subset X} \min_{\varphi: Z \to Y} \left(\sum_{z \in Z} ||z - \varphi(z)||^p + \sum_{y \in (Y \setminus \mathrm{im}(\varphi)) \cup X \setminus Z} \gamma(y)^p \right)^{\frac{1}{p}}.$$

We call the Bottleneck distance $B(X,Y) := V_\infty(X,Y)$. Since we now have three different Wasserstein distances, we want to make clear, that the function $V_p(\_,\_)$ will be the distance in question from now on.
In figure 9 an example of all the different distances, a single point from the set X has to consider is shown. We want to find a mapping that reduces the sum of the selected lengths, or the maximum such length respectively, with every vertex contributing to exactly one of these lengths.

3.1 Efficient Calculation

We quickly noticed that the whole problem can be modelled quiet easily via a min-cost-flow problem. This reduces the computations required from essentially checking exponentially many mappings to finding a min-cost-flow, which can be done in polynomial time.
So for $V_p(X,Y)$ we define the following graph. $V := X \cup Y \cup s, t, h, h'$ and $E := \big((X \cup h) \times (Y \cup h')\big) \cup \big(s \times (X \cup h)\big) \cup \big((Y \cup h') \times t\big)$.

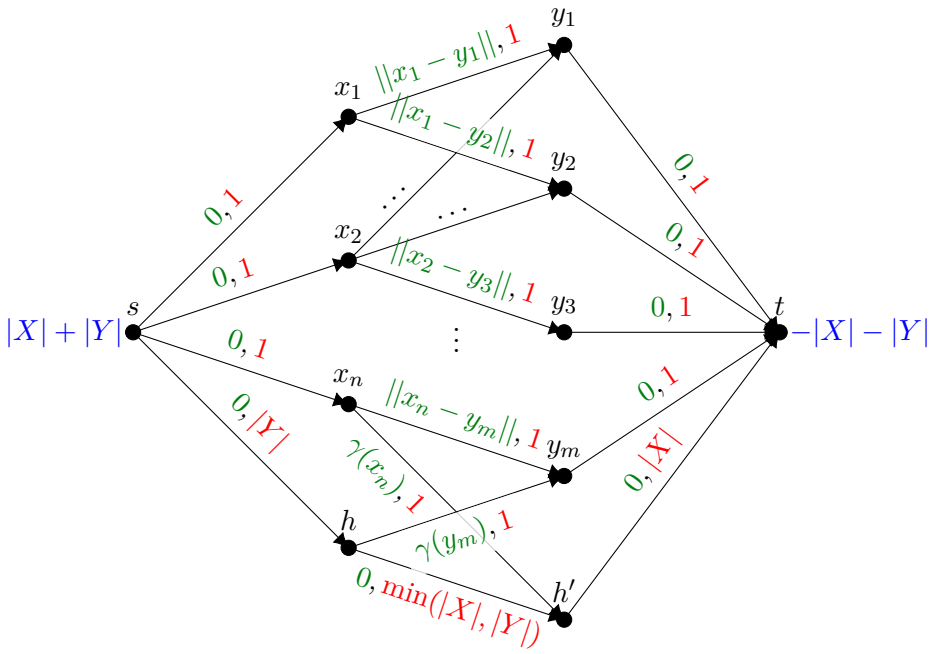

Fig. 10: Structure of the constructed graph. Flow requirements are blue, capacities red and costs green.

Weights are selected as follows. For edges in $(s \times (X \cup h)) \cup ((Y \cup h') \times t) \cup \{(h, h')\}$ they are always 0. For edges of the form $(x, y) \in X \times Y$ the costs are exactly $||x - y||$, and for edges of the form $(x, h') \in X \times \{h'\}$ resp. $(h, y) \in \{h\} \times Y$ we select $\gamma(x)$ or $\gamma(y)$. As capacities we choose $|Y|, |Y|$ and $\min(|X|, |Y|)$ for $(s, h), (h', t)$ and (h, h') respectively and 1 for all other edges. And finally the flow requirements $b(s) = -b(t) = |X| + |Y|$, and $b(v) = 0$ for all others. In figure 10 one can see the structure of the constructed graph.
In order to solve the Wasserstein distance via a min-cost flow instance, we must first prove that for every feasible integer flow there is a $Z \subset X$ and map $\varphi : Z \to Y$, where the cost of the flow is equal to the error term of φ and vice versa. Since the graph only has integral values, the min-cost-flow is integral as well, which is why these are sufficient to consider. From this it follows that the cost of a min-cost-flow is equal to the Wasserstein distance. Note that this cost only applies to $p = 1$. For $1 < p < \infty$ exponentiate the cost of all edges by p and return the pth root of the cost at the end.

Lemma 1. *The cost of a min-cost-flow for the above graph $G = (V, E)$ is equal to the Wasserstein distance $V_p(X, Y)$.*

Proof. Let $f : E \to \mathbb{Z}_{\geq 0}$ be a feasible flow. For Z choose all vertices x from X, such that $f((x, y)) = 1$ for a $y \in Y$ and set $\varphi(x) = y$ for this y. In less

mathy terms, φ is given by the selected flow edges of f, between X and Y. Since the cost of the edges of the form (x, h') or (h, y) selected by the flow is exactly $\gamma(x)$ and $\gamma(y)$, $V_p(X, Y)$ is a lower bound for the min-cost-flow value.
For the other direction we consider for given Z, φ the flow given by $f(x, y) = 1$ iff $x \in Z$ and$\varphi(x) = y$. For all $x \in X \setminus Z$ we set $f((x, h')) = 1$ and for $y \in Y \setminus \mathrm{im}(\varphi)$ we set $f((h, y)) = 1$. Thus one receives a feasible flow, with the same costs. □

And as we know, the min-cost-flow problem can be calculated in polynomial time with an algorithm like Edmonds-Karp or Dinic's.
To calculate the Bottleneck distance, we consider the subgraph given by all edges with weights smaller than a given value. Since the Bottleneck distance is given by the cheapest edge, so that with all cheaper edges a feasible flow is still possible, one can determine the Bottleneck distance with logarithmically many calls of a min-cost-flow algorithm with a procedure like binary search. This reduction follows from the fact, that the error term that is to be minimized in the Wasserstein distance is the same as the p-norm. The Bottleneck distance hence is the same as the ∞-norm, and is described by the maximum. Hence the Bottleneck distance reduces to finding the value

$$\begin{aligned}&\min_{\varphi: Z \subset X \to Y} \max\left(\max_{x \in Z} ||x - \varphi(x)||, \max_{y \in Y \setminus \mathrm{im}\varphi \cup X \setminus Z} \gamma(y)\right)\\ &= \min\left\{C \in \mathbb{R} \middle| (V, E_C) \text{ permits an } |\mathrm{X}|+|\mathrm{Y}| \text{ flow}\right\},\end{aligned}$$

where $E_C = \{e \in E | c(e) \leq C\}$. This leaves us with the following run time for both distances.
Let X and Y again be arbitrary. Then the graph consists of $n = 2+|X|+|Y|$ vertices and $m = 3 + 2|X| + 2|Y| + |X||Y|$ edges. Assuming a runtime of $O(n^2 m)$ for Dinic's algorithm, we have a runtime of $O(|X||Y|^3 + |X|^3|Y|)$ for the Wasserstein distance. For the Bottleneck distance we get a runtime of $O(\log(|X||Y|)(|X||Y|^3 + |X|^3|Y|))$. Note that Dinic's Algorithm is not necessarily the fastest for this type of problem, since the very easy structure of the graph opens the door to more specialized algorithms.

4 Feature Proposals

To combine all of this to actually calculate a distance of two images, we need to reduce an image to a point set. For this we looked at different types of feature detection algorithms and compared these.

```
public static List<PointD> sample(Mat matrix, int radius) {
    List<PointD> points = new ArrayList<>();
    // Create a "white" 2D point for each pixel above a
        given threshold
    ...
    List<PointD> samples = new ArrayList<>();
    while (!points.isEmpty()) {
        // Add a random white point to the samples and remove
            all white points within twice a given radius of
            the new sample
        int index = (int) (Math.random() * points.size());
        PointD sample = points.remove(index);
        samples.add(sample);
        points.removeIf(point ->
            point.subtract(sample).length() < 2 * radius);
    }
    ...
    return samples;
}
```

Fig. 11: *Codesnippet of the sampling algorithm*

4.1 Feature Detection

To do this we first considered an image, converted it to grayscale and then tried out existing feature detection algorithms. We tested the Laplace operator, the Harris operator and Canny edge detection.
The Laplace operator computes the 2nd spatial derivative and therefore may output many different gray values for one image, which is less usable for our approach. That is because we are trying to represent image structures with points, which is not possible to do in an accurate manner, if the feature detection outputs only a ”probability” for an edge or corner.
Next up is the harris operator, which highlights corners inside an image. One could think, that this is exactly what we need, but in fact we need to describe lines as series of points and not only start and end point. That rules out the Harris operator for our case as well.
Finally we tested the Canny operator, which only outputs either 0 or 1 for each pixel, given a threshold. Its used for edge detection and given the previous explanation exactly what we need. There might be some dynamic threshold computation left to do, so that the user doesn't have to choose that parameter on its own.

4.2 Feature Selection

Now we have a feature detection, that outputs a bitmap with ones for all edges in the image. The next thing to do is to convert that bitmap into

2D points, that will be used to compute the persistent homology. The idea behind the used algorithm is inspired by Poisson-Disc sampling and works as follows: We sample the area above the threshold by only sampling points, that are at least twice a given radius r away from all other samples (see fig. 11).
The result is a set of points, that resemble the original area, and we could put discs of radius r around each sampled point, without any discs intersecting. That way we have control over the granularity of the sampling using only a radius parameter. The given space between each sample ensures, that the numerical accuracy of the Voronoi algorithm does not have to be that exact.

Conclusion

In this work we have presented two different ways to compute the persistent homology of a given 2-dimensional point set. We began with a very naïve way, where we simply built up a big matrix, and continuously solved it for its kernel. We then used information from the kernel to reduce the size of the matrix and gather information about the persisten homology.
The second approach was much more involved, and made changes to the underlying graph, to keep the complexity of the problem small from the very beginning. For this we had to be careful, to respect the homology.
We then used the calculated persistent homology, to compute their difference. For this we changed the Wasserstein and Bottleneck distances to also accept differently sized point sets and made this change in a way, that it respects the significance of the underlying data - namely the persistent homology.
To compute this, we reduced the problem to a flow-instance, and showed, that the distances can be computed in polynomial time. Namely for $|X| = n$ and $|Y| = m$ we achieved a run time of $O(nm^3+n^3m)$ and $O(\log(nm)(nm^3+n^3m))$ respectively.
To put all of this to use, we presented some already well-known methods, to reduce an input image to a set of points, that represent the shape of the object in the image well. We also presented a method, of how to reduce the number of these point sets to increase the speed of our method without loosing much of the accuracy.
An important limitation however is, that this handcrafted approach really only works well with 2-dimensional data. It would be interesting to see, if there is a similar approach to the contractions on the graph for higher dimensional input data/a Čech complex with higher non-trivial homologies.
Another limitation is, that this distance is quite reliant on scale, since a scaled image produce scaled persistence diagrams. And since our Wasserstein distance version does not consider different scales, this produces big distances between scaled images.

Notes on the Java Program

The Java program is a maven project and therefore does not need any prerequisites but a java SDK of version 11+ and optionally a GNU plot installation. There are four different applications:

- DistanceApp: Allows the user to define two point sets and compute and visually display the Wasserstein and Bottleneck distances.
- FeatureApp: Allows the user to load images, compute feature sets using different selection algorithms and sample the result using disc sampling.
- HomologyApp: Allows the user to generate point sets and iteratively or completely compute and visualise the homology.
- MainApp: Allows the user to open two images and compute the Wasserstein and Bottleneck distance between them.

All applications can be executed from the source code. It is also possible to execute the package maven goal with any of the profiles `distance`, `feature`, `homology` or `main` active. Doing that generates a jar file inside the target folder, that can be executed using simply `java -jar [file]` with a java runtime. Each executable generated jar file ends with `jar-with-dependencies.jar`. You can find the executable jar files attached as well.

Here is a quick usage for each of those applications:
DistanceApp:
You can add points by left- or right-clicking with the mouse.
Keyboard shortcuts:

- Space: Compute distances
- Delete: Delete all points
- W: Toggle rendering of Wasserstein distance
- B: Toggle rendering of Bottleneck distance
- I: Load red points from file
- O: Load blue points from file
- P: Set specific p for the Wasserstein distance

FeatureApp:
Keyboard shortcuts:

- O: Open image from file
- L: Compute Laplace operator

- H: Compute Harris operator
- C: Compute Canny operator with the current threshold
- +: Increase threshold
- -: Decrease threshold
- Space: Sample the filtered image using disc sampling
- R: Reset to opened image
- Escape: Close the application

HomologyApp:
You can add points using left- and remove them using right-clicks. Existing points can be dragged using the left mouse button.
Keyboard shortcuts:

- Ctrl+O: Open a saved point file
- Ctrl+S: Save current point set to a file
- Ctrl+Shift+O: Import sampling from an image file
- Ctrl+Shift+S: Export persistence diagram to a file
- Escape: Close the application
- Space: Execute the next action
- Enter: Execute all (remaining) actions
- Ctrl+R: Reset cycles and actions
- Ctrl+Shift+C: Contract all new edges if possible (see 2.2.2)
- Delete: Remove all points
- Ctrl+Shift+R: Add 10 random points
- Ctrl+C: Toggle display of cycles (top left)
- Ctrl+A: Toggle display of actions (top right)
- Ctrl+D: Toggle display of Delaunay edges
- Ctrl+E: Toggle display of Voronoi edges
- Ctrl+V: Toggle display of Voronoi vertices
- Ctrl+Scroll-wheel: Increase/decrease vertex size (visualisation of persistence diagram)
- Ctrl+P: Plot persistence diagram using GNU plot (you will be asked to locate the gnuplot.exe)

MainApp:
You will be asked to open two image files and then the application computes and outputs the Wasserstein distance for $p = 1$ and the Bottleneck distance.

References

[Ghr14] Ghrist, Robert: *Elementary Applied Topology*. 1. CreateSpace Independent Publishing Platform, 2014. – ISBN 1502880857,9781502880857

[Hat02] Hatcher, Allen: *Algebraic topology*. Cambridge : Cambridge University Press, 2002. – ISBN 0–521–79160–X; 0–521–79540–0