Arnur Nigmetov, MSc

Comparison of Topological Summaries

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Betreuer:

Univ.-Prof. Dr.-Ing. Michael Kerber

Institut für Geometry

Graz, November 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral dissertation.

Date	Signature

Abstract

In this thesis we study the problem of computing distances between common descriptors of Topological Data Analysis (TDA).

First, we consider the classical case of 1-parameter persistence. Here a so-called barcode is a complete discrete invariant that captures all homological information at all scales. We provide algorithms to compute the bottleneck and the Wasserstein distances between barcodes. We also released an implementation of the algorithms; this software is now widely used in TDA community. The efficiency of the algorithms is verified by experiments.

Second, we consider the matching distance. We provide an algorithm and implementation to approximate the matching distance between 2-parameter persistence modules. The theoretical complexity of the algorithm is significantly better than the complexity of a recent algorithm (by Lesnick, Oudot and Kerber) that computes it exactly. We also demonstrate experimentally that our implementation makes computation of the matching distance feasible in practice.

Last, we consider a more abstract setting. Suppose that we need to work in a metric space such that an evaluation of a single distance is a costly operation. (examples: almost every metric space of topological summaries). We present a general strategy that allows us to approximate a matrix of pairwise distances between some set of points, reducing the number of distance computations.

Contents

Acknowledgements

First of all, I thank my adviser Michael Kerber. There is a countless number of reasons, including his scientific guidance, lectures, and support in all organizational matters.

I also thank Dmitriy Morozov for the opportunity to visit him for three summers in a row. This was a great experience, and I learnt a lot from the projects that we worked on, and from inspirational conversations about science and programming.

I am very grateful to Michael Lesnick and Ulrich Bauer for their consent to be reviewers of this thesis and the valuable remarks.

Johannes Wallner never refused to help me, both in his capacity of the head of the institute of geometry, and in private matters, like lending a drill. I think that every PhD student of Mathematics in Graz knows and likes Leonardo Alese, and it was a pleasure to share office with him, as well as with other students of Michael, Hannah and René. Of course, thanks go also to all other members of the institute, current and former, for the warm and friendly atmosphere.

Last, but not least, I thank my family, in particular, my wife Marina.

To my grandmothers, Rosa and Gemma (1932-2017).

1 Introduction

This thesis is about Topological Data Analysis (TDA). Before we give the outline of the following chapters, let us briefly discuss TDA in general. A usual way of explaining the main idea of topological data analysis is that it 'allows one to describe the shape of data with topological means'. We will generally refer to the objects that are computed in TDA as either *topological summaries* or *topological descriptors*.

Historically, TDA started with studying 1-parameter persistent homology, and the first descriptor was a *barcode*. This topological summary and other descriptors defined in its terms, such as persistence landscapes [28], are, probably, used most often in applications of TDA. The classical pipeline of 1-parameter persistent homology looks as follows.

1. Input: finite set S of points in $\mathbb{R}^n$.
2. For each r, consider a closed ball $B_r(p)$ of radius r around each point p of S, and build the space $X_r = \bigcap_{p \in S} B_r(p)$.
3. Replace the continuous nested family of spaces X_r with a finite nested family (*filtration*) of abstract simplicial complexes, namely, the nerves of the collection of balls $\{B_r(p) \mid p \in S\}$ for critical values of r, i.e., the values where the topology of X_r changes.
4. Apply a *homology* functor to the filtration of complexes. The result is an algebraic object called *persistence module*.
5. Compute the *barcode* of the persistence modules. A barcode is a finite multiset of intervals $[b_i, d_i)$. Each interval encodes the value of radius $r = b_i$ where a homological feature is born (a hole is formed in X_r), and the value of radius $r = d_i$ where the feature dies (the hole in X_r is filled). Draw the barcode in plane as the multiset of points (b_i, d_i). This graphical representation of a barcode is called a *persistence diagram*, and we will often use these terms as interchangeable.

Each of these steps poses many interesting questions. One can consider an arbitrary metric space instead of $\mathbb{R}^n$, or even a more general situation, as in [58], [57]. Computation of the nerve at step 3 can be too expensive or not suitable for a concrete problem, and we can replace the nerve (also called Čech complex) with another complexes, such as Vietoris-Rips, Delaunay, or α-complex. In step 4, we usually mean the homology with coefficients in some finite field. There exist generalizations to the case of integral homology [92]. The fact that the homological information is fully summarized in the form of a barcode can be generalized to persistence modules with infinite-dimensional vector spaces. The zigzag persistence works with a family of spaces X_i, in which the inclusion maps can go in either direction: $X_i \hookleftarrow X_{i+1}$ or $X_i \hookrightarrow X_{i+1}$ [35]. Since we want to apply these techniques to real data, the algorithmic aspects of all the aforementioned computations play an extremely important role. How can we compute the complexes? How can we compute the barcode? Is it possible to approximate the answer, if the input is too large? All these algorithmic problems were extensively studied, and we provide a short list of references in the next chapter.

The topic of this dissertation is related to the next phase: having computed a barcode, what can we do with it? Direct interpretation of a single persistence diagram is hard. However, in many scenarios we have multiple inputs (e.g., a set of 3D-shapes). Suppose that we found a distance (an extended metric) on the space of barcodes. The distance allows us to make *quantitative* statements about the similarity of these inputs. For example, we can formalize what we mean by *topological simplification*. Moreover, a matrix of pairwise distances can be fed into many machine learning or statistical algorithms. Last, but not least: all these steps can be performed by the user who knows very little about homology, but cares about a concrete problem related to real-world data.

Not all distances are equally useful. On the one hand, we want small perturbations of the input data sets to cause only small changes in the distance between their barcodes. On the other hand, we want to tell apart different inputs, and for this purpose it is better to have a distance that changes fast. A useful distance must provide some trade-off between these two properties. The two most popular distances on the space of barcodes are the *bottleneck* and the *Wasserstein* distances. The bottleneck distance is *universal*, i.e., the most discriminative among all stable distances (the adjectives 'stable' and

'discriminative' have a precise meaning here). The Wasserstein distance is stable in some weaker sense, but it is more discriminative. Roughly speaking, the bottleneck distance captures only the largest difference between all bars, while the Wasserstein distance sums the differences between the bars of the two barcodes. There are some other distances, though. The standard Hausdorff distance between sets in $\mathbb{R}^2$ is a stable, but rather weak distance, hence it is not so useful in practice. On the other hand, there are several different ways of linearizing the space of barcodes, that is, embedding barcodes in some Hilbert or Banach functional space. One such approach was already mentioned — persistence landscapes of Bubenik do exactly this. Another way of implicit embedding into Hilbert spaces is the famous kernel trick from machine learning. Obviously, any linearization defines its own distance, by the norm of the functional space. Normally one wants to have a stable embedding, hence these distances are also weaker than the bottleneck distance. See also [36] for some fundamental limitations on embeddings.

So far we only discussed 1-parameter persistence. There are applications that require studying the evolution of spaces indexed by more than one parameter, and multi-parameter persistence is currently a very active part of TDA. However, there are fundamental mathematical reasons that explain why this theory is significantly more difficult than in the 1-parameter case. In particular, a multi-parameter persistence module cannot be described by a single discrete invariant: there is no analogue of barcode. Therefore, it is the persistence module itself that serves as a topological summary in the multi-parameter setting.

The role played by the bottleneck distance in the 1-parameter case is taken by the *interleaving distance* — it is *universal* (the most discriminative among stable distances). The 1-parameter specialization of the interleaving distance is exactly the bottleneck distance. Unfortunately, even the problem of 3-approximating the interleaving distance for 2 parameters was proven to be NP-hard. Luckily, there exists a distance that is stable and can be computed in polynomial time — the *matching distance*. Roughly one can describe it as the maximal bottleneck distance (with some weighting) between 1-parameter projections of the persistence module on all lines which go in the positive direction. The universality of the interleaving distance implies that the matching distance gives a *lower* bound on the interleaving distance.

There also exists a *bottleneck distance for multi-parameter persistence modules*, which provides an *upper* bound on the interleaving distance [51].

Finally, let us talk about the applications of TDA. They are extremely diverse, and include neuroscience, medicine (classification of dicesases, drug design), cosmology, geology, shape analysis, etc. We refer the reader to the article [63]. The bibliography in this survey, published 2 years ago, contains 51 items, and most of the papers on this list are devoted to concrete applications of persistent homology to real data. If we agree that Reeb graphs and merge trees can be classified as topological summaries and hence belong to TDA, then the number of applications will, probably, increase by an order of magnitude (this is not completely fair, because merge trees and Reeb graphs had been studied in the context of visualization before the term TDA was coined).

This thesis considers different problems connected with computing distances between topological summaries, always aiming for implementations useful in practice, even if the algorithms only give an approximate answer. We guarantee an approximation ratio of $1 + \varepsilon$, so that the user has full control over the relative error.

The thesis is organized as follows.

Background

In Chapter 2, we collect the necessary definitions and recall the most important results which are directly related to the topic.

Bottleneck and Wasserstein distances

In Chapter 3, we work with the Wasserstein and bottleneck distances between persistence diagrams. Given two persistence diagrams X and Y, we define a complete bipartite graph by connecting each point of X with each point of Y (there is one very important, but more technical detail - we must add some points on the diagonal $b = d$, so that both partitions of the graph

have the same cardinality). This graph has a natural weighting induced by the ℓ_∞-distance in the (b, d)-plane, and, by construction, admits a perfect matching. The bottleneck cost of a perfect matching is the length of the longest edge; the q-Wasserstein cost is the sum of all lengths raised to the power q (q must be a real number from $[1, \infty)$). The bottleneck distance is the minimum of bottleneck costs over all perfect matchings, and the Wasserstein distance is the minimum of the Wasserstein costs (raised to power $1/q$, similarly for L_q-spaces). In order to compute the bottleneck distance, we use a variant of binary search, i.e., we implement the primitive $is\_less(X, Y, c)$, which returns `true` if and only if the bottleneck distance between X and Y is less than c. In other words, this primitive checks whether the graph has a perfect matching using edges of length at most c. Efrat et al. showed [60] how one can modify the Hopcroft-Karp algorithm for checking the existence of perfect matching for the case of geometric graphs, thus bringing down the complexity of this problem to $O(n^{1.5} log n)$. Instead of the complicated (and asymptotically optimal) geometric data structure used in their paper for Euclidean distances, we use k-d trees, the data structure that is known to perform well in practice.

The problem of Wasserstein distance computation is exactly the assignment problem. Many algorithms were proposed for solving it; there are two papers known to us that consider the assignment problem in the geometric context. We decided to use the auction algorithm proposed by Bertsekas. In this algorithm we consider the points of one diagram as *bidders* competing for the points of the other diagram (which in this context are often called *goods* or *items*). Each item has a price (a dual variable), the cost of an item for a bidder is the sum of its price and the cost (i.e., length) of the edge connecting the bidder and the item. We modify k-d trees to incorporate the weighting information, so that the search for the best item is performed efficiently.

The Hera software that implements our algorithms has been applied in practical problems, e.g., in physics [91], in action recognition [103], in medical problems [13]. It significantly outperformed the alternatives that had been available for distance computation when it was released, and now Hera is integrated into Dionysus 2 library.

Matching distance

In Chapter 4, we switch to bi-filtrations. Let us define the matching distance. Fix two bi-filtrations $X_{r,s}$ and $Y_{r,s}$. Consider an arbitrary line with positive slope in the (r,s)-plane. We can project the bi-filtrations onto this line, getting filtrations X^ℓ, Y^ℓ, and compute the bottleneck distance between them. The matching distance is defined as

$$\mathrm{d_M}(X,Y) := \sup_\ell w(\ell) W_\infty(X^\ell, Y^\ell).$$

Here $w(\ell)$ is the weight of the line ℓ. The weighting is necessary, because for lines with slope close to 0 or 1, the bottleneck distance will usually tend to infinity. It was shown recently [73] that the matching distance can be computed *exactly* in polynomial time. However, the exponent of this algorithm is rather high. The natural strategy to *approximate* the matching distance is to compute the expression $f(\ell) := w(\ell) W_\infty(X^\ell, Y^\ell)$ for some sample of lines ℓ. For sampling, we need to parameterize the set of lines with positive slope; we assume that the parameters vary over some axis-aligned rectangle. Then the most naive approach is to put some grid over the rectangle and compute $f(\ell)$ for all lines ℓ parameterized by the centers of the grid cells. This approach has two drawbacks. First, there is no guarantee of the relative error. Second, the computation of $f(\ell)$ is an expensive operation, which we would like to avoid whenever possible, and we expect that this straightforward algorithm would require a fine grid with many cells to produce a reliable answer. How can we guarantee the quality of our approximation? It is clear that $f(\ell)$ is continuous with respect to the standard topology on the set of lines. If we can quantify this statement, i.e., if we can find a function $\psi(\ell_1, \ell_2)$ such that

$$|f(\ell_1) - f(\ell_2)| < \psi(\ell_1, \ell_2), \tag{1.1}$$

then we can use the function ψ to compute the upper bound for the matching distance. The actual ψ will depend on the parameterization scheme that we choose; it is natural to expect that ψ will depend monotonically on the parameters and tend to 0 as lines $\ell_{1,2}$ approach each other. If we are able to provide a function ψ with these properties, then we can use it to compute the upper bound of the matching distance, if we only know the values of f at grid nodes. Equation (1.1) also provides the remedy against

the second issue of the naive algorithm. Instead of starting with some fine grid directly, we can use a hierarchy of grids. We start with a coarse grid and keep refining it until we reach the relative error specified by the user. When we need to refine a cell, it can happen that the value at its center is small enough, so that our estimate ensures that, for each line ℓ in this cell, $f(\ell)$ is less than the maximal value of f that we have computed so far. We conclude that there is no need to refine this cell, since it has no chance to improve the maximum.

This idea was applied to matching distance approximation in [20]. Our algorithm has the same structure, but we changed the parameterization scheme and performed a more accurate analysis of how $f(\ell)$ changes. The function ψ that is used in [20] is what we call a *global* bound: it only depends on the dimensions of the cell. We show that a *local* bound (a bound that takes into account the cell coordinates) can significantly accelerate the computation of $\mathrm{d_M}$ by reducing the number of computations of f.

Expensive metrics

The last chapter is motivated by two observations. First, none of the distances considered in this thesis can be computed in $O(1)$ time. If the input has size n, then the complexity of our algorithms for bottleneck, Wasserstein, and matching distances is polynomial; there is also a whole family of distances between merge trees and, more generally, Reeb graphs, which are NP-hard to compute. Secondly, we are normally interested not in computing a single distance between two given inputs, but, for a given set of n topological descriptors, we want to know all pairwise distances. The situation when the number of descriptors n is significantly smaller than the size of each of them is not uncommon, so it is reasonable to assume that the cost of one distance computation is much higher than any operation which is polynomial in n but does not involve a distance computation. We call this model a metric space with expensive distances. Let G be a graph with vertices $\{x_i\}$ from a metric space (X, ρ), to each edge (x_i, x_j) of G assign the weight $\rho(x_i, x_j)$ and let ρ_G denote the graph metric on G induced by the weighting. The graph G is called a *t-spanner*, if ρ_G is a t-approximation of

the original metric: $\rho_G(x_i, x_j) \leqslant t\rho(x_i, x_j)$ for all i, j. Our goal is to compute a spanner efficiently in the sense of our model, where computation of a single distance costs much more than other operations. Since we work in the general setting of an arbitrary metric space, the only tool that we have is triangle inequality. The idea is to use the distances that are known to us at the moment to get upper and lower bounds on the distances that have not yet been computed. We show experimentally that this strategy works well for approximating distances between persistence diagrams coming from a shape dataset, reducing both the number of distance computations and the running time even for the diagrams of moderate size.

We also consider an approximate nearest neighbor problem. Here we are given a set of points $P = \{p_1, \ldots, p_n\}$; as a preprocessing step we compute all pairwise distances between points of P. Given a query point q, we range over all points of P (in random order) and try to use the distances between p_i and q that we computed before to decide whether we need to compute the distance between q and the current point. Assuming doubling property, we can prove that in expectation we only need to compute $O(\log n)$ distances between q and points of P. We also run experiments that confirm these theoretical estimates.

2 Background

As explained in the previous chapter, the main objects for this thesis are distances between topological descriptors. In this chapter we collect the definitions of these descriptors and distances. There is no original work in this chapter, and most of the material can be found, for instance, in books [54], [65], [94].

Therefore we omit all proofs, only marking some statements that can be easily derived from definitions. Regarding generality, we choose the most general formulation, if it is not harder to state than a less general one; otherwise, we choose a more restrictive alternative.

2.1 Conventions and Notation

We assume the following notions to be known:

- Sets, equivalence relations, quotients.
- Groups, rings, fields, modules. Homomorphisms, isomorphisms and quotients of these structures.
- Vector spaces over arbitrary fields, linear maps, quotient spaces.
- Topological spaces, continuous maps.
- Metric spaces. Note that by metric we always mean an extended metric, i.e., all standard metric axioms must hold, but the metric can take value $+\infty$.

For a set X, we write id_X for the identity mapping ($x \mapsto x$ for all $x \in X$). We write 2^X for the power set of X, and $|X|$ for the cardinality of X. We use

square brackets for equivalence classes, if the equivalence relation $\sim$ is clear from the context:

$$[x] = \{y \in X \text{ such that } x \sim y\ \}.$$

$\mathbb{R}$ denotes the field of real numbers; $\mathbb{Z}$ denotes the ring of integers; $\mathbb{Z}/p$ denotes the field with p elements (p is prime). The symbol 0 can denote the number 0, the zero vector space over any field, the abelian group with one element, the zero vector in any vector space, and so on.

The sup-norm of a function $f\colon X \to \mathbb{R}$ is

$$\|f\|_\infty := \sup_{x \in X} |f(x)|.$$

If $x \in \mathbb{R}^d$, then $\|x\|_\infty$ denotes the ℓ_∞-norm:

$$\|(x_1, \ldots, x_d)\|_\infty := \max\{|x_i|\}.$$

We define the sup-norm for functions $f\colon X \to \mathbb{R}^d$, $x \mapsto (f_1(x), \ldots, f_d(x))$ by putting

$$\|f\|_\infty := \sup_{x \in X} \|f(x)\|_\infty = \sup_{x \in X} \max_i |f_i(x)|.$$

Let (X, ρ) be a metric space. $B_r(x)$ denotes the closed ball of radius r with center x:

$$B_r(x) = \{y \in X \mid \rho(x, y) \leqslant r\}.$$

Note that $B_r(x) = \varnothing$ for $r < 0$. $\mathbb{S}^n$ denotes the n-dimensional sphere in the Euclidean space.

2.2 Categories

In this thesis we do not need any deep results from category theory. However, in many situations a definition of some concrete objects, say, persistence modules, becomes more concise and natural, when phrased in the language of categories and functors. Categorification is also an important trend in TDA (see, for example, [48], [29], [10], or [8]). Therefore we include this short section and freely use categorical notions in the sequel.

Definition 2.1. *A* category $\mathbf{C}$ *consists of the following data:*

- *Class of objects (notation:* $\operatorname{Ob}\mathbf{C}$*).*
- *Class of morphisms (notation:* $\operatorname{Mor}\mathbf{C}$*).*
- *Mappings* $dom : \operatorname{Mor}\mathbf{C} \to \operatorname{Ob}\mathbf{C}$ *and* $cod : \operatorname{Mor}\mathbf{C} \to \operatorname{Ob}\mathbf{C}$*. For a morphism* f*, the object* $domf$ *is called the* domain *of* f*, and the object* $codf$ *is called the* codomain *of* f*. Notation: if* a *and* b *are the domain and codomain of* f*, then one writes* $f\colon a \to b$ *and says that* f *is a morphism from* a *to* b*. For each pair of objects* (a, b) *the class of morphisms from* a *to* b *must be a set, which is denoted* $\hom_C(a, b)$ *and is often called a* hom-*set.*
- *The composition law: for each triple of objects* a*,* b *and* c *there is a mapping* $\hom_C(b, c) \times \hom_C(a, b) \to \hom_C(a, c)$*, denoted by* $f, g \mapsto f \circ g$*. The composition law must be associative:* $(f \circ g) \circ h = f \circ (g \circ h)$ *whenever the composition is defined. A pair of morphisms* (f, g) *with* $f\colon b \to c$ *and* $g\colon a \to b$ *is called a* composable *pair.*
- *Identity morphisms. For each object* a*, the set* $\hom_C(a, a)$ *must contain a special element* id_a *that serves as the left and right identity with respect to composition:* $id_a \circ f = f$ *and* $h \circ id_b = h$*.*

Note that $\operatorname{Mor} C$ is a disjoint union $\cup_{a,b} \hom_C(a, b)$ over all objects a, b, so it is enough to specify the hom-sets. Normally one does not need to worry about the difference between classes and sets in this definition, but this is not always true: in [30] the authors show that the most general definition of persistence modules immediately leads to well-known problems.

Definition 2.2. *A morphism* $f\colon a \to b$ *is an* isomorphism*, if there exists a morphism* $g\colon b \to a$ *such that* $f \circ g = id_b$ *and* $g \circ f = id_a$*. Objects* a, b *are called* isomorphic*, if there exists an isomorphism* $f\colon a \to b$*.*

Definition 2.3. *A* functor *F from category* $\mathbf{C}$ *to category* $\mathbf{D}$ *consists of a mapping* $F\colon \operatorname{Ob}\mathbf{C} \to \operatorname{Ob}\mathbf{D}$ *and a mapping* $F\colon \operatorname{Mor}\mathbf{C} \to \mathbf{D}$ *such that the following conditions are satisfied:*

1. *If* $f \in \operatorname{Mor}_{\mathbf{C}}(a, b)$*, then* $F(f) \in \operatorname{Mor}_{\mathbf{D}}(F(a), F(b))$*.*
2. *F maps identity to identity:* $F(id_a) = id_{F(a)}$*.*
3. *F respects composition:* $F(f \circ g) = F(f) \circ F(g)$ *for each composable pair* $f, g \in \operatorname{Mor}\mathbf{C}$*.*

It is easy to prove that a functor takes isomorphisms to isomorphisms.

Definition 2.4. *A category* $\mathbf{C}$ *is a* subcategory *of category* D*, if* $Ob(C) \subseteq Ob(D)$*,* $Mor(C) \subseteq Mor(D)$*, and the composition of morphisms in* C *agrees with the composition in* D*, i.e.* $f \circ_C g = f \circ_D g$ *for all composable pairs of morphisms* $f, g \in Mor(C)$*.*

Definition 2.5. *Let* F, G *be functors* $\mathbf{C} \to \mathbf{D}$*. A* natural transformation *from* F *to* G *is a collection of morphisms* $\varphi(c) \hom_{Dcat}(F(c)G(c))$*, one for each object* c *of* $\mathbf{C}$*, such that for each morphism* $f \in \hom_{\mathbf{C}}(c_1, c_2)$ *we have* $G(f) \circ \varphi(c_2) = \varphi(c_1) \circ F(f)$*. If such a natural transformation exists, we write* $\varphi \colon F \dashrightarrow G$*.*

Examples. Category **Set**. The objects of this category are all sets. The set $\hom_{\mathbf{Set}}(a, b)$ is the set of all mappings from set a into set b, and the composition of morphisms is the usual composition of mappings.

A morphism in **Set** is an isomorphism exactly when it is a bijection. This example shows that one cannot replace the word 'class' in the definition of a category with the word 'set', since the class of all sets cannot be a set. The categories **Ab**, **Top**, and $\mathbf{vect}_{\mathbb{F}}$ (defined below) are subcategories of **Set**.

Category **Ab**. The objects of this category are all abelian groups, the set $\hom_{\mathbf{Ab}}(a, b)$ is the set of all homomorphisms from group a into group b. Isomorphisms in **Ab** are usual isomorphisms of the abelian groups (same is true for all categories of algebraic objects, such as group, rings, fields, modules over some fixed ring, etc).

Category **Top**. The objects of this category are all topological spaces, and morphisms are continuous maps between them. Isomorphisms in **Top** are called *homeomorphisms*. In TDA, we normally consider spaces that can be built from elementary blocks, say, cubes or simplices, using only a finite number of them; that is, we normally work in a subcategory of **Top**.

Category **TopPair**. An object of this category is a pair (X, A), where X is a topological space and A is a subspace of X. A morphism from (X, A) to (Y, B) is a continuous map $f \colon X \to Y$ such that $f(A) \subseteq B$.

Category **HTop**. Let us first recall the definition of homotopy.

Definition 2.6. *Let X and Y be topological spaces, and f, g be continuous maps $X \to Y$. A* homotopy *between f and g is a continuous map $H\colon X \times [0,1] \to Y$ such that $H(x,0) = f(x)$ and $H(x,1) = g(x)$ for all $x \in X$. If a homotopy between f and g exists, they are said to be* homotopic*, and we write $f \sim g$.*

Informally, f and g are homotopic, if f can be continuously deformed into g. It is easy to prove that $\sim$ is an equivalence relation, which is compatible with composition in the following sense: if $f_{1,2}\colon X \to Y$ and $g_{1,2}\colon Y \to Z$ are continuous maps such that $f_1 \sim f_2$ and $g_1 \sim g_2$, then $g_1 \circ f_1 \sim g_2 \circ f_2$. This means that we can define composition of the equivalence classes by putting $[f] \circ [g] := [f \circ g]$ for $f\colon Y \to Z$ and $g\colon X \to Y$. We obtain the category **HTop** whose objects are topological spaces and whose morphisms are not mappings, but equivalence classes of mappings (checking all axioms of category is straightforward). The identity morphism of a space X is, of course, the homotopy class of the identity map $[id_X]$. Isomorphisms in this category play an important role in topology and are called *homotopy equivalences*. They can be described directly, without referring to categorical notions.

Definition 2.7. *Let X and Y be topological spaces, f be a continuous map $X \to Y$, and g be a continuous map $Y \to X$. We say that the pair (f,g) is a* homotopy equivalence *between spaces X and Y, if $f \circ g \sim id_Y$ and $g \circ f \sim id_X$. Two topological spaces are* homotopy equivalent*, if there exists a homotopy equivalence (f,g) between them. Each of the mappings f,g can also be called a homotopy equivalence.*

Category $\mathbf{vect}_{\mathbb{F}}$. Let $\mathbb{F}$ be a field. The objects of $\mathbf{vect}_{\mathbb{F}}$ are finite-dimensional vector spaces over $\mathbb{F}$, the morphisms are $\mathbb{F}$-linear maps.

Poset as category. Recall that a relation $\preceq$ on a set X is a partial order, if it is reflexive ($a \preceq a$), antisymmetric ($a \preceq b$ and $b \preceq a$ imply $a = b$), and transitive ($a \preceq b$ and $b \preceq c$ imply $a \preceq c$). Given a partially ordered set (*poset*) $(X, \preceq)$, we define the category $\mathbf{X}$ by setting $\operatorname{Ob}\mathbf{X} = X$. If elements a and b are not comparable, then we put $\hom_{\mathbf{X}}(a,b) = \varnothing$, else we put $\hom_{\mathbf{X}}(a,b)$ to contain a single element. The composition of morphisms is well-defined, because $\preceq$ is transitive. The only isomorphisms in $\mathbf{X}$ are the identity morphisms. These categories also play a significant role in TDA.

Examples of functors that are relevant to TDA: *homology* and *persistence module*. They will be defined in next sections.

2.3 Simplicial Complexes

Definition 2.8. *The set of points* $\{p_1, \ldots, p_n\}$ *in* $\mathbb{R}^N$ *is* affinely independent*, if the set of vectors* $\{p_2 - p_1, \ldots, p_n - p_1\}$ *is linearly independent.*

Obviously, a subset of an affinely independent set is affinely independent. Note that the empty set and the set of cardinality 1 are affinely independent by definition.

Definition 2.9. *A* geometric simplex *T of dimension* $d \geqslant 0$ *is a convex hull of* $d+1$ *affinely independent points* $V = \{v_0, \ldots, v_d\}$ *in* $\mathbb{R}^n$*. Every point* v_i *is called a* vertex *of the simplex. A geometric simplex defined by a subset of V is called a* face *of T.*

Examples: a 0-dimensional simplex is a single point, a 1-dimensional simplex is a line segment between two points, a 2-dimensional simplex is a triangle (with its interior), etc. We can use geometric simplices as building blocks, glueing them together along faces to obtain new spaces.

Definition 2.10. *A set of geometric simplices* $K = \{\sigma_i | i \in I\}$ *is a* geometric simplicial complex*, if*

1. *for each* $\sigma_i \in K$*, all faces of* σ_i *belong to K, and*
2. *for all* $i, j \in I$ *the intersection* $\sigma_i \cap \sigma_j$ *is a face of both* σ_i *and* σ_j*.*

We say that K is a triangulation *of* $X = \cup_{i \in I} \sigma_i$*. A simplex* $\sigma \in K$ *is* maximal*, if it is not a face of any other simplex from K.*

We may sometimes abuse the terminology by referring to X itself as a geometric simplicial complex, if it is clear from the context, which triangulation is meant. We always assume that geometric simplicial complexes are finite. The first condition in this definition implies that a geometric simplicial complex is fully determined by its maximal simplices.

For example, let us take 4 points $a, b, c, d \in \mathbb{R}^3$ such that the triangles $\{a, b, c\}$, $\{a, b, d\}$, $\{a, c, d\}$, and $\{b, c, d\}$ intersect only along their common edges. We define a geometric simplicial complex K by taking these 4 triangles as its maximal simplices. The union of the triangles is the boundary of the tetrahedron spanned by $\{a, b, c, d\}$. From the topological point of view, we constructed a 2-dimensional sphere by taking 4 triangles and glueing them along their edges. Most importantly, in order to identify this complex as a sphere, we only needed to know which vertices span a simplex in this space. This motivates the definition of abstract simplices and abstract simplicial complexes.

Let V be a fixed set, which we assume to be finite. We refer to elements of V as vertices.

Definition 2.11. *An* abstract simplex of dimension *k on a set of vertices V is a subset of V of cardinality $k + 1$. Simplex τ is a* face *of simplex σ, if $\tau \subseteq \sigma$. An* abstract simplicial complex *on V is a set of simplices K such that if $\sigma \in K$, then all faces of the simplex σ belong to K.*

Note that K must be finite, since $|V| < \infty$. If we remove from an abstract simplicial complex K all simplices of dimension greater than m, the remaining simplices form a subcomplex of K. This subcomplex is called the *m-skeleton* of K. Simplices of dimension 1 are called *edges* of K.

Definition 2.12. *Let K_1 and K_2 be two abstract simplicial complexes with vertex sets V_1 and V_2. A mapping $f\colon K_1 \to K_2$ is a* simplicial map, *if a) for each $v_1 \in V_1$ considered as a 0-dimensional simplex of K_1, its image $f(v_1)$ is a 0-dimensional simplex in K_2 and b) for each $\sigma = \{v_1, \dots, v_{d+1}\} \in K_1$ its image $f(\sigma)$ must be equal to $\{f(v_1), \dots, f(v_{d+1})\} \in K_2$.*

Thus a simplicial map is completely defined by its restriction onto vertices. Notice that different vertices of K_1 can be mapped to the same vertex of K_2, so a d-dimensional simplex is mapped to a simplex of dimension at most d. All simplicial complexes and simplicial maps between them form a category **Simp**. If there exists an isomorphism between two complexes, we say that these complexes are *isomorphic*, or have the same combinatorial structure, or are combinatorially equivalent.

A geometric simplicial complex $\{\sigma_i\}$ can be viewed as an abstract simplicial complex K: take V to be the union of all vertices of all σ_i-s and add an abstract simplex τ to K if and only if the geometric simplex spanned by the vertices of τ is one of the σ_i-s.

Definition 2.13. *A geometric simplicial complex $X = \cup_{i\in I}\sigma_i$ is a* geometric realization *of an abstract simplicial complex K, if K is isomorphic to X viewed as abstract simplicial complex.*

It is easy to prove that a) every (finite) abstract simplicial complex K has a geometric realization, and b) all geometric realizations of K are homeomorphic to each other. This allows us to use abstract simplicial complexes in context where one expects a topological space.

As we observed, every geometric simplicial complex yields an abstract simplicial complex. Here is another way of building abstract complexes.

Definition 2.14. *Let $\mathfrak{U} = \{U_\alpha \mid \alpha \in A\}$ be a finite collection of sets. Its nerve $\mathcal{N}(\mathfrak{U})$ is an abstract simplicial complex on A defined as follows:*

$$\mathcal{N}(\mathfrak{U}) = \{\sigma \mid \bigcap_{\alpha\in\sigma} U_\alpha \neq \varnothing\}.$$

In words: a simplex $\sigma = \{\alpha_0, \ldots, \alpha_d\}$ belongs to $\mathcal{N}(\mathfrak{U})$ if and only if the intersection of the corresponding elements U_{α_i} of $\mathfrak{U}$ is not empty.

Typically the sets U_α are subspaces of some topological space. The following theorem, which is usually called the *nerve lemma*, plays a very important role in TDA.

Theorem 2.15. *Suppose that each $U_\alpha \in \mathfrak{U}$ is a closed and convex subset of $\mathbb{R}^d$. Then the nerve $\mathcal{N}(\mathfrak{U})$ is homotopy equivalent to $\bigcup_\alpha U_\alpha$.*

This theorem allows us to replace a continuous object, which can be very complicated from the computational point of view, with a simple combinatorial object, the nerve, without losing the information about the homotopy type. On the other hand, the dimension of the nerve can be much higher than d. There are other variations of the nerve lemma. For example, one can require U_α to be open and all intersections to be contractible (homotopy equivalent to a point).

2.4 Homology

Let $\mathbb{F}$ be either the ring of integers $\mathbb{Z}$ or some field, K be an abstract simplicial complex. By definition, its group of d-chains $C_d(K, \mathbb{F})$ is a free $\mathbb{F}$-module generated by all d-simplices of K. In other words, a d-chain in K with coefficients in $\mathbb{F}$ is a formal linear combination of d-dimensional simplices of K with coefficients from $\mathbb{F}$, and $C_d(K, \mathbb{F})$ is the set of all d-chains with operations defined in a natural way. There is one important technicality here. For each simplex $\sigma \in K$, we fix some ordering of its vertices, $\sigma = [v_0, \dots, v_d]$. Let s be a permutation of $\{0, \dots, d\}$, and $\varepsilon(s)$ be its sign. If we rearrange the vertices of σ, then we put $[v_{s(0)}, \dots, v_{s(d)}] := (-1)^{\varepsilon(s)}\sigma$. Note that if $1 = -1$ in $\mathbb{F}$, then the order of vertices does not matter. We define the boundary of σ as the $(d-1)$-chain

$$\partial\sigma := \sum_{k=0}^{d} (-1)^k [v_0, \dots, \hat{v_k}, \dots, v_d],$$

where $\hat{v_k}$ means that the vertex v_k is omitted. We extend the mapping ∂ to all chains by linearity:

$$\partial(\sum \lambda_i \sigma_i) := \sum \lambda_i \partial\sigma_i.$$

Thus, for each dimension $d > 0$, we have an $\mathbb{F}$-linear map $\partial\colon C_d(K, \mathbb{F}) \to C_{d-1}(K, \mathbb{F})$. We are going to suppress K and $\mathbb{F}$ for brevity, and as long as there is no ambiguity, C_d is just a shorthand notation for $C_d(K, \mathbb{F})$. Let

$$Z_d = Z_d(K, \mathbb{F}) := \ker \partial\colon C_d \to C_{d-1} \text{ for } d > 0$$

and

$$B_d = B_d(K, \mathbb{F}) := \operatorname{im} \partial\colon C_{d+1} \to C_d \text{ for } d \geqslant 0$$

The elements of Z_d are called *cycles*, and the elements of B_d are called *boundaries*. Zero-dimensional cycles are an exceptional case: here we set $Z_0 := C_0$. Thus $Z_d \subseteq C_d$ and $B_d \subseteq C_d$ for all $d \geqslant 0$. The most fundamental fact is that every boundary is a cycle, i.e., $B_d \subseteq Z_d$. The proof amounts to a straightforward verification of $\partial \circ \partial = 0$.

Definition 2.16. *The* d-th homology group of K with coefficients in $\mathbb{F}$ *is the quotient*

$$H_d = H_d(K, \mathbb{F}) := Z_d / B_d.$$

Note that $H_d(K, \mathbb{F})$ depends only on $(d+1)$-skeleton of K. We write $[c] \in H_d$ for the equivalence class of a cycle $c \in Z_d$ and call it a *homology class* of c. Recall that the boundary operator was defined for some fixed orientation of simplices; it can be proven that $H_d(K, \mathbb{F})$ does not depend on this choice. If $\mathbb{F}$ is a field, then Z_d and B_d are finite-dimensional vector spaces, therefore H_d is also a finite-dimensional vector space. If $\mathbb{F}$ is $\mathbb{Z}$, then Z_d and B_d are free finitely-generated abelian groups, and H_d is a finitely-generated, but not necessarily free abelian group.

Let $f\colon K_1 \to K_2$ be a simplicial map. We extend it to chains (keeping the same symbol f) by setting

$$f(\sum \lambda_i \sigma_i) := \sum \lambda_i f(\sigma_i).$$

We define $H_d(f)\colon H_d(K_1) \to H_d(K_2)$ by $H_d(f)([c]) := [f(c)]$. One can check that $H_d(f)$ is well-defined. Now we can say that $H_d(K, \mathbb{F})$ is functorial in K. If $\mathbb{F} = \mathbb{Z}$, then $H_d(\cdot, \mathbb{Z})$ is a functor $\mathbf{Simp} \to \mathbf{Ab}$, and if $\mathbb{F}$ is a field, then $H_d(\cdot, \mathbb{F})$ is a functor $\mathbf{Simp} \to \mathbf{vect}_{\mathbb{F}}$. In case $\mathbb{F} = \mathbb{Z}$, the abelian group H_d can be computed by bringing the matrix of the boundary operator ∂ to Smith normal form. For field coefficients we need to perform Gaussian elimination, which is much easier from the computational point of view. Unfortunately, this simplification comes at a price: the homology of X with field coefficients contains less information about the topology of X. More precisely, if we know $H_d(X, \mathbb{Z})$ for all d, then we can determine $H_d(X, \mathbb{F})$ for any field $\mathbb{F}$ (but not vice versa) via the *universal coefficient theorem*.

We defined H_d for abstract simplicial complexes. Let X be a *triangulable* topological space (i.e., X is a geometric realization of some abstract simplicial complex K). We define the *simplicial homology* $H_d(X, \mathbb{F})$ of the space X with coefficients in $\mathbb{F}$ to be $H_d(K, \mathbb{F})$. The fact that $H_d(X, \mathbb{F})$ is well-defined (does not depend on the choice of triangulation K) is not easy to prove. In fact, simplicial homology is not the best choice for mathematical purposes, because a) the class of triangulable spaces is too small, even if one allows infinite simplicial complexes (for instance, not all topological manifolds are triangulable) and b) as we just remarked, the topological invariance of simplicial homology is relatively hard to establish. Nevertheless, we choose this definition, because it is simple and well-suited for algorithmic purposes. Simplicial complexes and simplicial maps are easy to represent in the computer.

There are other ways to define homology of a topological space (not necessarily triangulable), but all these definitions give the same result when applied to triangulable spaces. One of these variants is called *singular homology*. It is used in most textbooks on algebraic topology, and we do not define it here. We only remark that singular homology with integral coefficients is actually a functor $\mathbf{HTop} \to \mathbf{Ab}$. This compact formulation implies that:

- If X and Y are homotopy equivalent, then they have the same homology in all dimensions: $H_d(X) \cong H_d(Y)$ for all $d \geqslant 0$ (in particular, spaces of different dimension can have the same homology).
- If maps $f, g\colon X \to Y$ are homotopic, then they induce the same map in homology.
- If $f\colon X \to Y$ is a homotopy equivalence, then the induced map in homology $H_d(f)\colon H_d(X) \to H_d(Y)$ is an isomorphism.

All these statements also hold for simplicial homology.

Relative homology and cohomology. Let $A \subseteq X$ be a subcomplex of X. We say that a chain $c \in C_d(X)$ is a *relative cycle*, if its boundary belongs to $C_{d-1}(A)$, and write $Z_d(X, A)$ for the set of d-dimensional relative cycles. We say that c is a *relative boundary*, if there exists a chain $c' \in C_d(A)$ such that $c - c'$ is a boundary, and write $B_d(X, A)$ for the set of relative boundaries. The *relative homology group* $H_d(X, A; \mathbb{F})$ of the pair (X, A) is defined as the quotient $Z_d(X, A)/B_d(X, A)$. Again, it would be more accurate to call $H_d(X, A)$ the *simplicial* relative homology group. There exists a *singular* relative homology, which is defined for arbitrary topological spaces. We mention the relative homology theory here because it is used in one of the variants of TDA, so-called *extended persistence* [43].

We can dualize the chain groups, i.e., consider $\mathbb{F}$-linear maps from chains into $\mathbb{F}$. The space of such maps is called the group of cochains. The boundary operator determines its dual, the *coboundary operator* d that goes in the opposite direction, from cochains of dimension n to cochains of dimension $n + 1$. We define cocycles and coboundaries in the same way as we did for cycles and boundaries. Since the identity $d \circ d = 0$ holds, the quotient group is well-defined, and it is called the *cohomology group*. Apart from its

importance in pure mathematics, cohomology is also useful in TDA, see [47].

2.5 Filtrations

Recall that we regard $\mathbb{R}^d$ as a poset with $(a_1, \dots, a_d) \preceq (b_1, \dots, b_d)$ if and only if $a_i \leqslant b_i$ for all $i = 1 \dots d$.

Definition 2.17. *A multi-filtration of a topological space X is a map $a \mapsto X_a$ that associates to each $a \in \mathbb{R}^d$ a subset X_a of X such that $X_a \subseteq X_b$ whenever $a \preceq b$. A multi-filtration of a simplicial complex K is a map $a \mapsto K_a$ that associates to each $a \in \mathbb{R}^d$ a subcomplex K_a of K such that $K_a \subseteq K_b$ whenever $a \preceq b$. For a simplex σ, every minimal element of the closure of the set $\{a \in \mathbb{R}^d \mid \sigma \in K_a\}$ is called a* critical value *of σ in the multi-filtration. [1] If all simplices in K have only one critical value, the multi-filtration is called* 1-critical.

We always assume that all spaces X_a are triangulable. If $d = 1$, we omit the prefix multi and just say *filtration*, or even *monofiltration*. Note that all monofiltrations are 1-critical.

Definition 2.18. *We call a multi-filtration of spaces (respectively, complexes) X_a* nice, *if*

1. *For all $a \in \mathbb{R}^d$ and for all sufficiently small ε, the inclusion $X_a \to X_{a+\varepsilon}$ is a homotopy equivalence (respectively, identity).*
2. *There exist finitely many numbers $c_1, \dots, c_N$ with the following property. If $a = (a_1, \dots, a_d)$ satisfies $a_i \neq c_j$ for all i, j, then for all sufficiently small ε, the inclusion $X_{a-\varepsilon} \to X_a$ is a homotopy equivalence (respectively, identity).*

We always assume that multi-filtrations are nice. Note that this definition is not standard, and is rather restrictive. However, it is sufficient for the purposes of this thesis: the most popular classes of multi-filtrations usually satisfy these requirements, and they provide sufficiently many examples of how topological descriptors can be obtained. Nice multi-filtrations are

[1] Usually this set is closed and has only finite number of minimal elements.

obviously discrete objects that can be easily represented in the computer. We can think of them as finite grids formed by simplicial complexes which are connected by inclusion maps. Additionally, we need to store the real coordinates associated to the grid nodes.

Examples. *Čech complexes*. Let $S = \{x_1, \ldots, x_n\}$ be a finite set of points in a metric space, and let X_r be the union of closed balls of radius r centered at x_i,

$$X_r := \cup_i B_r(x_i).$$

One can prove that $\{X_r\}$ is a nice filtration.

Definition 2.19. *The nerve of the set $\{B_r(x_i)\}$ is called the* Čech complex of S *and denoted $\check{C}_r(S)$. In other words,*

$$\check{C}_r = \{\sigma \in 2^S \mid \exists y \in \mathbb{R}^d \text{ such that } \|y - x_i\| \leqslant r \text{ for all } x_i \in \sigma\}.$$

Obviously, $r \mapsto \check{C}_r(S)$ is a filtration of abstract simplicial complexes. By theorem 2.15, $\check{C}_r(S)$ is homotopy equivalent to X_r. Let σ be a subset of S. The critical value of σ in $\check{C}_r$ is the diameter of the minimal enclosing ball of σ.

Vietoris-Rips complexes.

Definition 2.20. *The* Vietoris-Rips complex *with radius r of the set S is the abstract simplicial complex VR_r with the set of vertices S given by the following rule:*

$$VR_r(S) = \{\sigma \in 2^S \mid \operatorname{diam}(\sigma) \leqslant 2r\}.$$

It is known [54] that $\check{C}_r(S) \subseteq VR_r(S) \subseteq \check{C}_{\sqrt{2}r}(S)$ (in $\mathbb{R}^n$ with Euclidean distance). Thus Vietoris-Rips complexes can be interpreted as approximation of the Čech complexes. The advantage of the corresponding filtration is that we only need to consider pairwise distances between elements of σ; for higher-dimensional simplices we are greedy, adding them to the complex once we have all their edges.

Sublevel-set multi-filtration. Let $f\colon X \to \mathbb{R}^d$ be a function on a triangulable space X. For $y \in \mathbb{R}^d$, we put

$$SL_y = SL_y(X, f) := \{x \in X \mid f_i(x) \leqslant y_i \text{ for all } i = 1, \dots, n\}$$

Obviously, $y \mapsto SL_y$ is a multi-filtration of topological spaces. Again, we assume that this multi-filtration is nice (this is true, for instance, if f is a piecewise-linear map). The Čech filtration is a special case of the sublevel-set multi-filtration: put f to be the distance from its argument to S.

Lower-star multi-filtration. Let K be an abstract simplicial complex on a set of vertices V, and let $\varphi\colon V \to \mathbb{R}^d$ be some function. For $y \in \mathbb{R}^d$, we put

$$LS_y(K) := \{\sigma \in K \mid \varphi_i(v) \leqslant y_i \text{ for all } v \in \sigma, \text{ for all } i = 1, \dots, d\ \}$$

The filtration $y \mapsto LS_y(K)$ is called the *lower-star multi-filtration* of K defined by the function φ.

2.6 Persistence Modules and Interleaving Distance

What happens if we apply the homology functor to a multi-filtration? We obtain a family of vector spaces indexed by the points of $\mathbb{R}^d$; these vector spaces are connected by the linear maps induced by the inclusion maps of the multi-filtration. Such objects are called persistence modules, and in this section we define them formally.

Let us fix some field $\mathbb{F}$; as usual, $\mathbf{vect}_{\mathbb{F}}$ denotes the category of finite-dimensional vector spaces over $\mathbb{F}$.

Definition 2.21. *A* persistence module *in d parameters is a functor M from $\mathbb{R}^d$ (regarded as a poset category) to $\mathbf{vect}_{\mathbb{F}}$. If a and b are elements of $\mathbb{R}^d$ such that $a \preceq b$, we write $M(a \preceq b)$ for the linear map $M(a) \to M(b)$. These linear maps are called* structure maps. *A* morphism *φ from persistence module M_1 to M_2 is a natural transformation $M_1 \dashrightarrow M_2$. In other words, φ is a collection of linear*

maps $\varphi(p)\colon M_1(p) \to M_2(p)$ *such that for each* $p \preceq q$ *the following diagram commutes*

$$\begin{array}{ccc} M_1(p) & \xrightarrow{M_1(p \preceq q)} & M_1(q) \\ \downarrow{\scriptstyle \varphi(p)} & & \downarrow{\scriptstyle \varphi(q)} \\ M_2(p) & \xrightarrow{M_2(p \preceq q)} & M_2(q) \end{array}$$

Persistence modules form a category, and a morphism φ is an isomorphism in categorical sense if and only if φ is a point-wise isomorphism (for all $p \in \mathbb{R}^d$, the linear map $\varphi(p)$ is an isomorphism). By zero module we mean the module that assigns the zero vector space to each $p \in \mathbb{R}^d$. We denote the zero module by 0.

Definition 2.22. *A* direct sum *of persistence modules* M_1 *and* M_2 *is defined point-wise:* $(M_1 \oplus M_2)(p) := M_1(p) \oplus M_2(p)$*, and* $(M_1 \oplus M_2)(q \preceq p) := M_1(q \preceq p) \oplus M_2(q \preceq p)$*. A module* M *is called* indecomposable*, if* $M \cong M_1 \oplus M_2$ *implies* $M_1 \cong 0$ *or* $M_2 \cong 0$*.*

Theorem 2.23. *Every persistence module* M *is isomorphic to a direct sum of indecomposable modules:*

$$M \cong \bigoplus_{\alpha \in A} M_\alpha,$$

modules M_α *are uniquely (up to isomorphism and re-indexing) determined by* M*.*

This theorem is a special case of the Krull-Schmidt-Azumaya theorem. A proof can be found, e.g., in [25].

Definition 2.24. *An* interval *in a poset* $(P, \preceq)$ *is a non-empty subset* $I \subseteq P$ *that satisfies the following property: for all* $a, c \in I$ *and for all* $b \in P$ *such that* $a \preceq b \preceq c$*, the element* b *also belongs to* I*.*

Let I be an interval in $\mathbb{R}^d$. We define a corresponding *interval module* M_I as follows:

$$M_I(p) = \begin{cases} \mathbb{F}, \text{ if } p \in I, \\ 0, \text{ otherwise.} \end{cases}$$

$$M_I(a \preceq b) = \begin{cases} id_{\mathbb{F}}, \text{ if } a \in I \text{ and } b \in I, \\ 0, \text{ otherwise.} \end{cases}$$

It is easy to check that M_I is indecomposable. However, an indecomposable module need not be an interval module. Actually, there exist indecomposable modules that are not even *thin*, i.e., at some points the corresponding vector space has dimension greater than 1.

Our definition of a persistence module is rather restrictive: a persistence module in our sense is finite-dimensional at each point. A more general definition replaces $\mathbb{R}^d$ with an arbitrary poset and allows infinite-dimensional vector spaces. Studying persistence modules in this generality is a subject in its own even for 1-parameter case, see books [94], [39].

Fix a real $\varepsilon \geqslant 0$ and let $\boldsymbol{\varepsilon}$ denote the point $(\varepsilon, \dots, \varepsilon)$ of $\mathbb{R}^d$. We define a *shift functor* $S(\varepsilon)$ as follows. For a persistence module M, we put $S(\varepsilon)(M)(p) := M(p + \boldsymbol{\varepsilon})$ and $S(\varepsilon)(M)(p \preceq q) := M(p + \boldsymbol{\varepsilon} \preceq q + \boldsymbol{\varepsilon})$. For a morphism $\varphi\colon M_1 \to M_2$, we set $S(\varepsilon)(\varphi)(p) := \varphi(p + \boldsymbol{\varepsilon})$. It is easy to see that $S(\varepsilon)$ is indeed a functor.

Definition 2.25. *Let M be a persistence module, ε be a non-negative real number. The* ε-transition morphism *φ^{ε}_M is a morphism $M \to S(\varepsilon)(M)$ defined by $\varphi^{\varepsilon}_M(p) := M(p \preceq p + \boldsymbol{\varepsilon})$. Two persistence modules M_1 and M_2 are* ε-interleaved, *if there exist morphisms $\psi_1\colon M_1 \to S_\varepsilon(M_2)$ and $\psi_2\colon M_2 \to S_\varepsilon(M_1)$ such that $S(\varepsilon)(\psi_1) \circ \psi_2 = \varphi^{2\varepsilon}_{M_2}$ and $S(\varepsilon)(\psi_2) \circ \psi_1 = \varphi^{2\varepsilon}_{M_1}$. The* interleaving distance *between M_1 and M_2 is the infimum of ε such that M_1 and M_2 are ε-interleaved:*

$$D_I(M_1, M_2) = \inf\ \varepsilon \geqslant 0 \textit{ s. t. } M_1 \textit{ and } M_2 \textit{ are eps-interleaved.}\} \tag{2.1}$$

One can prove that D_I is a distance. The infimum in the definition of D_I is actually attained, as proved in [78]. In particular, if $D_I(M_1, M_2) = 0$, then the modules M_1 and M_2 are 0-interleaved, i.e., isomorphic.

Definition 2.26. *A distance d on the space of persistence modules is called* stable, *if for all functions $f_{1,2}\colon X \to \mathbb{R}^n$, and for all $m \geqslant 0$*

$$d(H_m(SL(f_1)), H_m(SL(f_2))) \leqslant \|f_1 - f_2\|_\infty$$

Recall that $SL(f_i)$ denotes the sublevel filtration.

The following theorem (proven in [78]) states that the interleaving distance is the most discriminative among all stable distances (for prime fields).

Theorem 2.27. *1. D_I is stable for all fields $\mathbb{F}$.*
2. If $\mathbb{F}$ is $\mathbb{Q}$ or $\mathbb{Z}/p$, and d is a stable distance, then $d(M_1, M_2) \leqslant D_I(M_1, M_2)$ for all M_1, M_2.

The fact that $H_m(\cdot, \mathbb{F})$ is functorial allows us to construct a persistence module from a given multi-filtration. This module depends on dimension m and on the field of coefficients $\mathbb{F}$. Nevertheless, we will abuse the terminology by referring to a distance between any of these modules as 'the distance between the two given multi-filtrations', because we will consider m and $\mathbb{F}$ being fixed (say, by the user).

2.7 1-Parameter Persistence

In 1-parameter case, the theory of persistence becomes much simpler, because we can explicitly describe all indecomposable modules.

Theorem 2.28. *If a 1-parameter persistence module M is indecomposable, then M is isomorphic to an interval module, $M \cong M_I$ for some I.*

Intervals in $\mathbb{R}$ can be closed, open, or half-open. We are usually interested in persistence modules that are obtained by applying homology to a nice filtration, hence we can assume that the intervals are of the form $[b, d)$, where d can be ∞. Thus, we only consider persistence modules M that can be decomposed as $M \cong \oplus_{j \in J} M_{[b_j, d_j)}$, where $b_j \in \mathbb{R}$ and $d_j \in \mathbb{R} \cup \{\infty\}$.

Definition 2.29. *Each interval $[b_j, d_j)$ is called a* bar, *and the multiset of all intervals $[b_j, d_j)$, $j \in J$ is called the* barcode *of the module M. A* persistence diagram *of M is a multiset that contains all points on the diagonal (b, b) with infinite multiplicity, and points (b_j, d_j) with the same multiplicity that the bar $[b_j, d_j)$ has in the interval decomposition of M.*

We call a multiset of points in $\mathbb{R}^2$ a *persistence diagram*, if this multiset consists of all points of the diagonal $\{(b,b) \mid b \in \mathbb{R}\}$ with infinite multiplicity and of finitely many points from the upper half-plane $\{(b,d) \in \mathbb{R}^2 \mid b < d\}$, each with finite multiplicity. The latter part of a diagram is called the *off-diagonal part*.

Definition 2.30. *Let X, Y be two persistence diagrams, let η be a bijection $\eta\colon X \to Y$; the supremum of ℓ_∞ distances between $x \in X$ and its image under η is called the* bottleneck cost *of η. The* bottleneck distance *between X and Y is the infimum of the bottleneck cost over all bijections $\eta\colon X \to Y$:*

$$W_\infty(X,Y) = \inf_\eta \sup \|x - \eta(x)\|_\infty.$$

Intuitively, we are trying to match the points of X and Y in such a way that minimizes the maximal distance between matched points. If we want to minimize not only the maximal distance, but their total sum, we get 1-Wasserstein distance. Analogously to L_q, we can generalize this definition by taking the sum of distances raised to the power of q, where q is a fixed real number from $[1,\infty)$.

Definition 2.31. *The q-*Wasserstein distance *between X and Y is*

$$W_q(X,Y) = \inf_\eta \left(\sum_{x \in X} \|x - \eta(x)\|_\infty^q \right)^{\frac{1}{q}}.$$

By the bottleneck or Wasserstein distance between 1-parameter persistence modules we mean the corresponding distance between the persistence diagrams of these modules; same for filtrations. Again, the latter case is a slight abuse of the language, because it depends on dimension in which we take the homology and the field of coefficients, but this should not lead to any confusion.

While it is straightforward to verify that W_∞ and W_q define extended metrics on the set of persistence diagrams, it is not immediately obvious that they are even computable, because the definitions involve bijections between multi-sets of cardinality continuum. However, a standard lemma, which is

explained in the next chapter, reduces the computation of W_∞ and W_q to the problem of finding an optimal matching in a finite graph.

The following theorem was proven in [78].

Theorem 2.32. *The interleaving distance between* 1*-parameter persistence modules is equal to the bottleneck distance between their persistence diagrams.*

In particular, the bottleneck distance is stable. Clearly, the Wasserstein distance can exceed the bottleneck distance, hence W_q cannot be stable by theorem 2.27. Nevertheless, the Wasserstein distance is stable in some weaker sense, see [44]. Some simple properties of the W_q and W_∞ are summarized in the following lemma, the first 2 items of which hold for $q \in [1, \infty]$.

Lemma 2.33.
1. *$W_{[}q]$ is shift-invariant: if we replace each point (p_1, p_2) in the diagrams X and Y with the point $(p_1 + a, p_2 + a)$, the distance does not change.*
2. *$W_{[}q]$ is homogeneous: for fixed $\lambda > 0$, if we replace each point (p_1, p_2) in the diagrams X and Y with the point $(\lambda p_1, \lambda p_2)$, the distance is multiplied by λ.*
3. $\lim_{q\to\infty} W_{[}q](X, Y) = W_\infty(X, Y)$.

In particular, if we take two mono-filtrations and add the same constant to the critical value of each simplex, the bottleneck distance between them does not change. If we multiply each critical value with the same $\lambda > 0$, the distance is also multiplied by λ.

Matching distance. Let $M_{1,2}$ be two 2-parameter persistence modules. For each line L with positive slope in $\mathbb{R}^2$, fix a point $p_L \in L$. Let $\vec{e}_L = (\cos\gamma, \sin\gamma)$ be the unit vector that is parallel to L and has positive coordinates. We identify L with $\mathbb{R}$ as follows: the Euclidean metric in plane induces the metric on L, p_L plays the role of 0, and the positive direction on L is the direction of $\vec{e}_L$. The restrictions of $M_{1,2}$ on L become 1-parameter persistence modules. We denote them by $restr(M_i)$.

Definition 2.34. *The* matching distance *between M_1 and M_2 is*

$$\mathrm{d_M}(M_1, M_2) := \sup_L w(L) W_\infty(restr(M_1), restr(M_2)),$$

where L ranges over all lines with positive slope, and $w(L) := \min(\cos\gamma, \sin\gamma)$.

d_M is well-defined: if we replace the point p with any other point of L, the bottleneck distance will not change by Lemma 2.33. The matching distance is stable and provides lower bound for the interleaving distance [77].

2.8 Reeb Graphs and Merge Trees

Let f be a continuous real-valued function on a triangulable space X.

Definition 2.35. *We define two points $x, y \in X$ to be equivalent, $x \sim y$, if $f(x) = f(y)$ and both x and y belong to the same connected component of the level set $f^{-1}(c)$ (here $c = f(x) = f(y)$). Function f factors through the canonical projection $\pi\colon X \to X/\sim$: there exists a unique $\bar{f}\colon X/\sim \to \mathbb{R}$ such that $f = \bar{f} \circ \pi$. The pair $(X/\sim, \bar{f})$ is called the Reeb graph of f.*

We also refer to the quotient space $X/\sim$ as the Reeb graph of f. It can be proven that Reeb graphs are really graphs (i.e., 1-dimensional simplicial complexes), if a) X is a smooth manifold and f is a so-called simple Morse function or b) X is a compact polyhedron and f is piecewise-linear, see [53]. If we replace level sets with sublevel sets, we get the definition of a merge tree:

Definition 2.36. *We define two points $x, y \in X$ to be equivalent, $x \sim y$, if $f(x) = f(y)$ and both x and y belong to the same connected component of the sublevel set $f^{-1}((-\infty, c])$ (here $c = f(x) = f(y)$). The quotient space $X/\sim$ (or the pair $(X/\sim, \bar{f})$, where $\bar{f}$ is obtained by factoring f through the canonical projection) is called the* merge tree *of f.*

We write $\mathcal{R}(f)$ for the Reeb graph of f and $\mathcal{MT}(f)$ for the merge tree of f. Note that the Reeb graph depends only on the 2-skeleton of the domain, hence it loses all information about homology in dimensions greater than 1. On the other hand, the Reeb graph and merge tree contain strictly more information than the barcode in dimension 0: it is straightforward to read off the persistence diagram from these descriptors, and two functions with different merge trees can have equal 0-dimensional persistence diagrams.

Let us define the *interleaving distance* between merge trees, introduced by Morozov, Beketayev and Weber. Following [83], we extend a merge tree with an arc from the root to $+\infty$. First, we define a *shift map* of the merge tree. Let $\varepsilon \geqslant 0$ and define $i^\varepsilon \colon \mathcal{MT}[f] \to \mathcal{MT}[f]$ as

$$i^\varepsilon([(x,y)]) := \text{connected component of } \bar{f}^{-1}((x,y+\varepsilon)) \text{ which contains } [(x,y)]$$

Geometrically, i^ε is moving points in $\mathcal{MT}_f$ by ε upwards (i.e., to the greater values of $\bar{f}$). Now let $\mathcal{MT}[f]$ and $\mathcal{MT}[g]$ be two merge trees, let ε be a positive number, and let $i^{2\varepsilon} \colon \mathcal{MT}[f] \to \mathcal{MT}[f]$ and $j^{2\varepsilon} \colon T_g \to T_g$ corresponding 2ε-shift maps.

Definition 2.37. *Two continuous maps $\alpha \colon \mathcal{MT}[f] \to \mathcal{MT}[g]$ and $\beta \colon \mathcal{MT}[g] \to \mathcal{MT}[f]$ are ε-compatible, if*

$$\hat{g}(\alpha(x)) = \hat{f}(x) + \varepsilon, \qquad \hat{f}(\beta(x)) = \hat{g}(x) + \varepsilon \tag{2.2}$$

$$\beta \circ \alpha = i^{2\varepsilon}, \qquad \alpha \circ \beta = j^{2\varepsilon} \tag{2.3}$$

The interleaving distance $d_I(\mathcal{MT}[f], \mathcal{MT}[g])$ *is defined as*

$$\inf\{\varepsilon : \exists \varepsilon\text{-compatible } \alpha \colon \mathcal{MT}[f] \to \mathcal{MT}[g], \beta \colon \mathcal{MT}[g] \to \mathcal{MT}[f]\} \tag{2.4}$$

In [2], the authors prove that it is NP-hard to approximate the interleaving distance between merge trees with a factor better than 3 (they use this result as a lemma to prove a similar hardness result for Gromov-Hausdorff distance). There are several other distances on the space of Reeb graphs. The paper [5] introduced the *functional distortion distance* between Reeb graphs and proved that it is equal to the interleaving distance when restricted to merge trees. In [48], the authors use the language of (co)sheaf theory and category theory to define Reeb graphs and the interleaving distance between them. This generalized interleaving distance is equivalent to the functional distortion distance (proved in [11]). The papers [38], [64] consider intrinsic metrics on the Reeb graphs. Finally, the *edit distance* between Reeb graphs is singled out by the property of being universal ([8]).

2.9 Algorithmic Aspects

Barcodes (persistence diagrams). Computation of 1-parameter persistence is very well studied. Suppose that we are given a filtered simplicial complex K_t, $K_t \subseteq K_{t+1}$. We index its simplices $\{\sigma_i \mid i = 1 \dots N\}$ in a way that is compatible with the filtration and dimension: if $i < j$, then neither can the simplex σ_i have dimension greater than dimension of σ_j, nor can σ_i appear in the filtration earlier than σ_j (formally, $\min\{t \mid \sigma_i \in K_t\} \leqslant \min\{t \mid \sigma_j \in K_t\}$). The boundary operator ∂ in this case can be written as a block matrix D, and computation of the barcode boils down to a special form of Gaussian elimination of this matrix, when columns are added from left to right only (one exception is an output-sensitive algorithm proposed in [40], which computes only bars of length exceeding some user-defined threshold). The first algorithms for computing the barcode appeared in [55] by Edelsbrunner, Letscher, and Zomorodian, (only for subcomplexes of S^3) and in [107] by Zomorodian and Carlsson. Several important algorithmic optimizations, data structures, and parallel and distributed algorithms were suggested in [41], [6], [47], [24]. In particular, computations can be significantly accelerated by combining the clearing optimization with computation of persistent cohomology instead of homology. Forman's discrete Morse theory can also be helpful in persistence computation.

While the theoretical worst-case complexity of most of these algorithms is cubic (and this is tight, [82]), they perform quite well in practice. There is also an algorithm that runs in matrix multiplication time $O(N^\omega)$, [86], but it seems to be more of theoretical interest. There are many implementations available now. An incomplete list of them includes:

- JavaPlex;
- Perseus (based on discrete Morse theory approach);
- CHomP;
- PHAT (shared-memory parallel implementation);
- DIPHA (distributed computation of persistence);
- GUDHI;
- Dionysus;
- Eirene;

- Ripser (the fastest software for computing the barcode using Vietoris-Rips filtration).

Dimension 0 is a special case, where one only needs to track the connectivity information. Not surprisingly, the algorithms here are based on the UnionFind data structure, which is very efficient. Therefore, in dimension 0 it is possible to process extremely large data sets, producing diagrams with millions of points.

Reeb graphs and merge trees. Computation of these topological summaries is also an extensively developed area, which is explained by the large role that they play in scientific visualization. For Reeb graphs, the state of the art algorithm is the one by Parsa [87]. For merge trees, we only mention the classical approach based on Kruskal's minimum-spanning-tree algorithm. Most of the algorithms in this area have the worst-case theoretical complexity of $O(n \log n)$, where n is the total number of edges, faces, and triangles. They also perform well in practice. As for implementations, we can mention the Topology Tool Kit, ttk. On the other hand, computation of distances between these descriptors is *in statu nascendi*. As far as I know, the only paper that has some positive (fixed-parameter tractability) results in this realm of NP-hardness is [61].

Multi-parameter persistence. Working with multi-parameter persistence is much harder than in the 1-parameter case. The paper [34], which introduced the notion, contained algorithms that are based on Gröbner basis. The interleaving distance was proven to be NP-hard to approximate in [23] (part of the reduction used in the proof in [23] appeared before in [22]). The RIVET software [100] can process different kinds of input, e.g., computing the function-Rips bi-filtration from a given data set. RIVET can visualize 2-parameter persistence modules, compute their minimal presentations, the matching distance and Hilbert function.

2.10 What is Omitted?

The main purpose of this chapter is to be a glossary for the following chapters, not to be a survey of TDA. We did not even mention many topics that are not necessary for definitions of W_∞, D_I, $\mathrm{d_M}$, and W_q. Perhaps, the most glaring omission is the interpretation of persistence modules as graded modules over polynomial rings and presentations of persistence modules. The reason is that a proper explanation of these notions would make the background section too large. On the other hand, the algorithm given in Chapter 4 generalizes to presentations of persistence modules in a straightforward way, and no algorithmic details are lost, if we only consider bi-filtrations. A short and incomplete list of other omitted topics:

- Further examples of filtrations and multi-filtrations, such as α-complexes, witness complexes, Delaunay filtrations ([54], Chap. III).
- We can also build spaces from cubes, not simplices. The cubical variant of *singular* homology goes back to J.-P. Serre's PhD thesis [96]. The homology of cubical spaces was also studied in the applied context. In particular, the *ring* structure of cubical cohomology is considered in [71]. Persistent cubical homology can be computed by an algorithm from [104]. See also the book [70]. Cellular homology applies to spaces built from cubes, but here we have more relaxed conditions that the maps attaching cubes to each other must satisfy. It may be necessary to work with CW-complexes even if the original problem is phrased for simplicial complexes [56].
- Persistence modules as quiver representations, Gabriel's theorem ([94]).
- Zigzag persistence [35];
- Generalizations and variations of the stability theorem (see [9], [26], [21] and the book [39]);
- Connections to Forman's discrete Morse theory ([80]);
- Connections to (co)sheaf theory ([45], [46]);
- Mapper was introduced in [97] and studied in [49], [37], [50].
- Further applications of TDA, including applications of persistence in pure mathematics (spectral geometry, symplectic geometry, see [90], [88], [89]).
- Statistical aspects of TDA.

3 Efficient Computation of Bottleneck and Wasserstein Distances

The material of this chapter first appeared in the ALENEX paper [74], and the extended version was published in the Journal of Experimental Algorithms [72].

3.1 Introduction

We defined the bottleneck and Wasserstein distances in Chapter 2. As we mentioned in Chapter 1, their computation is reduced to finding optimal matchings in a graph whose vertices are points in plane, and the weight of the edges is induced by some norm in $\mathbb{R}^2$. This metric structure leads to asymptotically improved algorithms that take advantage of data structures for near-neighbor search. This line of research dates back to Efrat et al. [60] for the bottleneck distance and Vaidya [102] for the 1-Wasserstein case. Rich literature has developed since then, mainly focusing on approximation algorithms for Euclidean metrics in low and high dimensions; see [1] for a recent summary.

Our contributions. Our contribution is two-fold. First, we provide an experimental study illuminating the advantages of exploiting geometric structure in assignment problems: we compare mature implementations of bottleneck and Wasserstein distance computations for the geometric and

purely combinatorial versions of the problem and demonstrate that exploiting the spatial structure improves running time and space consumption for the matching problem. Second, by focusing on the setup relevant in topological data analysis, we provide the fastest implementation for computing distances between persistence diagrams, significantly improving the implementation in the Dionysus library [81]. The latter prototypical implementation was the only publicly available software for the problem. Given the importance of this problem in applications, our implementation is therefore addressing a real need in the community. Our code is publicly available.This chapter contains the following specific contributions:

- For bottleneck matchings, we follow the approach of Efrat et al. [60]: they augment the classical combinatorial algorithm of Hopcroft and Karp [68] with a geometric data structure to speed up the search for vertices close to query points. We do not implement their asymptotically optimal but complicated approach. We instead use a k-d tree data structure [69] to prune the search for matching vertices in remote areas (also proposed by the authors). As expected, this strategy outperforms the combinatorial version that linearly scans all vertices. Several careful design choices are necessary to obtain this improvement; see Section 3.3.
- For Wasserstein matchings, we implement a geometric variant of the *auction algorithm*, an approximation algorithm by Bertsekas [15]. We use *weighted* k-d trees, again with the goal to reduce the search range when looking for the best match of a vertex. A data structure similar to ours appears in [4]. Our implementation outperforms a version of the auction algorithm that does not exploit geometry, which we implement for comparison, both in terms of running time and space consumption. Both our implementations of the auction algorithm dramatically outperform Dionysus, albeit computing approximations rather than the exact answers as the latter. Dionysus uses a variant of the Hungarian algorithm [85]; see Section 3.4.
- We extend our auction implementation to the case of points with multiplicities, or masses. While this problem can be trivially reduced to the previous one by replacing a multiple point with a suitable number of simple copies, it is more efficient to handle a point with multiplicity as one entity, splitting it adaptively only when fractions

are matched to different points. An extension of the auction algorithm to this case has been described by Bertsekas and Castañon [18]. We refer to it as *auction with integer masses*. Our implementation exploits the geometry of the problem in a similar way as the auction for simple points. Handling masses imposes a certain overhead that slows down the computation if the multiplicities are low. However, our experiments show that the advantage of the auction with integer masses becomes apparent already when the average multiplicity is around 10, and the performance gap between the two variants of the auction increases when the average multiplicity increases; see Section 3.5.

3.2 Background

Assignment problem. Given a weighted bipartite graph $G = (A \sqcup B, E, w)$, with $|A| = n = |B|$ and a weight function $w : E \to \mathbb{R}_+$, a *matching* is a subset $M \subseteq E$ such that every vertex of A and of B is incident to at most one edge in M. These vertices are called *matched*. A matching is *perfect* if every vertex is matched; equivalently, a perfect matching is a matching of cardinality n; it can be expressed as a bijection $\eta : A \to B$.

For a perfect matching M, the *bottleneck cost* is defined as $\max\{w(e) \mid e \in M\}$, the maximal weight of its edges. The *q-Wasserstein cost* is defined as $(\sum_{e \in M} w(e)^q)^{1/q}$; for $q = 1$, this is simply the sum of the edge weights. A perfect matching is *optimal* if its cost is minimal among all perfect matchings of G. In this case, the *bottleneck* or *q-Wasserstein cost* of G is the cost of an optimal matching. If a graph does not have a perfect matching, its cost is infinite. For $q > 1$, the q-Wasserstein cost can be reduced to the case $q = 1$ with the following simple observation.

Proposition 3.1. *The q-Wasserstein cost of $G = (A \sqcup B, E, w)$ equals q-th root of the 1-Wasserstein cost of $G' = (A \sqcup B, E, w^q)$, where w^q means that all edge weights are raised to the q-th power.*

We call a graph $G = (A \sqcup B, E, w)$ *geometric*, if there exists a metric space (X, d) and a map $\varphi : A \sqcup B \to X$ such that for any edge $e = (a, b) \in E$, $w(e) = d(\varphi(a), \varphi(b))$. In this case, we generally blur the distinction between

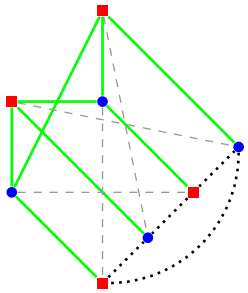


Figure 3.1: An example of G for two persistence diagrams with 2 off-diagonal points each. Skew edges are dashed gray, edges connecting diagonal points are dotted black.

vertices and their embedding and just assume for simplicity that $A \sqcup B \subset X$. The motivating example of this work is $X = \mathbb{R}^2$ and $d(x, y) = \|x - y\|_\infty$.

Persistence distance as a matching problem. Recall that persistence diagrams consist of finitely many off-diagonal points with finite multiplicity and all the diagonal points with infinite multiplicity. The task of computing $W_*(X, Y)$ can be reduced to a bipartite graph matching problem; we follow the notation and argument given in [54, Ch. VIII.4]. Let X_0, Y_0 denote the off-diagonal points of X and Y, respectively. If $u = (x, y)$ is an off-diagonal point, we denote its orthogonal projection on the diagonal $((x + y)/2, (x + y)/2)$ as u', which is the closest point to u on the diagonal. Let X_0' denote the set of all projections of X_0, that is $X_0' = \{u' \mid u \in X_0\}$. With Y_0' defined analogously as $\{v' \mid v \in Y_0\}$, we define $U = X_0 \cup Y_0'$ and $V = Y_0 \cup X_0'$; both have the same number of points. We define the weighted complete bipartite graph, $G = (U \sqcup V, U \times V, c)$, whose weights are given by the function

$$c(u, v) = \begin{cases} \|u - v\|_\infty & \text{if } u \in X_0 \text{ or } v \in Y_0 \\ 0 & \text{otherwise} \end{cases}. \tag{3.1}$$

Points from U and V are depicted as squares and circles, respectively, in Figure 3.1 on the left; all the diagonal points are connected by edges of weight 0 (plotted as dotted lines). The following result is stated as the *Reduction lemma* in [54, Ch. VIII.4]:

Lemma 3.2.

- $W_\infty(X, Y)$ *equals the bottleneck cost of* G.
- $W_q(X, Y)$ *equals the* q*-Wasserstein cost of* G*. This is equal to the* q*-th root of the* 1*-Wasserstein cost of* G^q*, which is the graph* G *with cost function* c^q*, raising all edge costs to the* q*-th power.*

Note that G is almost geometric: distances between vertices are measured using the L_∞-metric, except that points on the diagonal can be matched for free to each other if they are not matched with off-diagonal points. Can this almost-geometric structure speed up computation? This question motivates our work.

It is possible to simplify the above construction. We call an edge $uv \in U \times V$ a *skew edge* if $u \in X_0$, $v \in X_0'$ and v is not the projection of u, or if $v \in Y_0$, $u \in Y_0'$ and u is not the projection of v (skew edges are shown with dashed lines in Figure 3.1).

Lemma 3.3. *For both bottleneck and Wasserstein distance, there exists an optimal matching in* (G^q, c^q) *that does not contain any skew edge.*

Proof. Fix an arbitrary matching M and define the matching M' as follows: For any $uv \in M \cap X_0 \times Y_0$, add uv and $u'v'$ to M'. For any skew edge ab' of M with a the off-diagonal point (either in X_0 or Y_0), add aa' to M'. Also add to M' all edges of M of the form aa', where a is an off-diagonal point. It is easy to see that M' is a perfect matching without skew edges, and its cost is not worse than the cost of M: indeed, the skew edge ab' got replaced by aa' which is not larger, and the vertices on the diagonal possibly got rearranged, which has no effect on the cost. □

Lemma 3.3 implies that removing all skew pairs does not affect the result of the algorithm, saving roughly a factor of two in the size of the graph.[1]

We prove another equivalent characterization of the optimal cost which will be useful in Section 3.5: The previous lemma showed that, conceptually, increasing the weight of each skew edge to ∞ does not affect the cost of an optimal matching. We show now that even decreasing the weight of a skew

[1] DIONYSUS uses the same simplification.

edge ab' to the weight of aa' has no effect on the optimal cost. Formally, let us define $\tilde{G} = (U \sqcup V, U \times V, \tilde{c})$ with a new weight function $\tilde{c}$ as follows:

$$\tilde{c}(u,v) = \begin{cases} \|u - v\|_\infty & \text{if } u \in X_0 \text{ and } v \in Y_0 \\ \|u - u'\|_\infty & \text{if } u \in X_0 \text{ and } v \in X_0' \\ \|v - v'\|_\infty & \text{if } u \in Y_0' \text{ and } v \in Y_0 \\ 0 & \text{otherwise} \end{cases}. \tag{3.2}$$

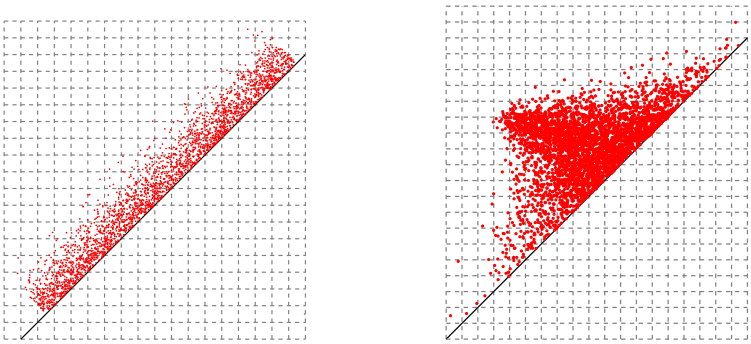
Lemma 3.4. *For both bottleneck and Wasserstein distance, there exists an optimal matching in $\tilde{G}$ that does not contain any skew edge.*

Proof. The proof of lemma 3.3 carries over word by word. □

Lemma 3.5. *The weighted graphs G and $\tilde{G}$ have the same bottleneck and Wasserstein cost.*

Proof. Let C be the cost for G, and $\tilde{C}$ be the cost for $\tilde{G}$ with respect to bottleneck or Wasserstein distance. Since $\tilde{c} \leqslant c$ edge-wise, $\tilde{C} \leqslant C$ is immediate. For the opposite direction, fix a matching $\tilde{M}$ that realizes $\tilde{C}$ and has no skew edge (such a matching exists by Lemma 3.4). By the absence of skew edges, the cost $\tilde{M}$ is the same if the cost function $\tilde{c}$ is replaced by c. This implies $C \leqslant \tilde{C}$. □

K-d trees. K-d trees [69] are a classical data structure for near-neighbor search in Euclidean spaces. The input point set is split into two halves at the median value of the first coordinates. The process is repeated recursively on the two halves, cycling through the coordinates used for splitting. Each node of the resulting tree corresponds to a bounding box of the points in its subtree. The boxes at any given level are balanced to have roughly the same number of points. Given a query point q, one can find its nearest neighbor (or all neighbors within a given radius) by traversing the tree. A subtree can be eliminated from the search if the bounding box of its root node lies farther from the query point than the current candidate for the nearest neighbor (or the query radius). Although the worst case query performance is $O(\sqrt{n})$ in the planar case, k-d trees perform well in practice and are easy to implement. In Section 3.3 we use the ANN [84] implementation



(a) Example of a *normal* diagram. (b) Example of a *real* diagram.

Figure 3.2: Examples of persistence diagrams.

of k-d trees, changing it to support the deletion of points. For Section 3.4 we implemented our own version of k-d trees to support the search for a nearest neighbor with weights.

Experimental setup. All experiments in this chapter were performed on a server running Debian wheezy, with 32 Intel Xeon cores clocked at 2.7GHz, with 264 GB of RAM. Only one core was used per instance in all our experiments.

We experimentally compare the performance both on artificially generated diagrams as well as on realistic diagrams obtained from point cloud data. For brevity, we restrict the presentation to two classes of instances. In the first class, we generate pairs of diagrams, each consisting of n points. The points are of the form $(a - |b|/2, a + |b|/2)$ where a is drawn uniformly in an interval $[0, s]$, and b is chosen from a normal distribution $N(0, s)$, with $s = 100$. In this way, the persistence of a point, $|b|$, is normally distributed, so the point set tends to concentrate near the diagonal. This matches the behaviour of persistence diagrams of realistic data sets, where points with high persistence are sparse, while the noise present in the data generates the majority of the points, with small persistence. For every set of parameters, we generate 10 independent pairs of diagrams. We refer to this class of experiments as *normal* instances (Figure 3.2(a)). To get a diagram of the second class, we sample a point set P of n points uniformly at random from either a 4-, or a 9-dimensional unit sphere. The 1-dimensional persistence

diagram of the Vietoris–Rips filtration of P serves as our input. We use the DIPHA library[2] for the generation of these instances. Note that persistence diagrams generated in this way have different numbers of points. We refer to this class of experiments as *real* instances (Figure 3.2(b)). For each set of parameters (sphere dimension and number of points sampled), we have generated 6 test instances and computed pairwise distances between all $\binom{6}{2} = 15$ pairs.

Our plots show the average running times and the standard deviation as error bars. For the *real* class, the x-axis is labelled with the number of points sampled from the sphere, not with the size of the diagram. The size of the persistence diagrams, however, depends practically linearly on the number of sample points, with a constant factor that grows with dimension: the largest instance for dimension 9 is a diagram with 5762 points, while for dimension 4 the largest diagram is of size 1679.

Our experiments cover many other cases. We tested various choices of s, the scaling parameter in the *normal* class, and of the sphere dimension in the *real* class. We also tried different ways of generating diagrams, for instance, by choosing n points uniformly at random in the square $[0, s] \times [0, s]$, above the diagonal. In all these cases, we encountered the same qualitative difference between the tested algorithms as for the two representative cases discussed in this chapter.

3.3 Bottleneck matchings

Our approach follows closely the work of Efrat et al. [60], based on the following simple observation. Let $G[r]$ be the subgraph of G that contains the edges with weight at most r. The bottleneck distance of G is the minimal value r such that $G[r]$ contains a perfect matching. Since the bottleneck cost for G must be equal to the weight of one of the edges, we can find it exactly by combining a test for a perfect matching with a binary search on the edge weights.

[2] `https://github.com/DIPHA/dipha`

The algorithm by Hopcroft and Karp. Efrat et al. modify the algorithm by Hopcroft and Karp [68] to find a maximum matching. We briefly summarize the Hopcroft–Karp algorithm; [60] provides an extended review. For a given graph $G[r]$, the algorithm computes a maximum matching, i.e., a matching of maximal cardinality. $G[r]$, with $2n$ vertices, has a perfect matching if and only if its maximum matching has n edges.

The algorithm maintains an initially empty matching M and looks for an *augmenting path*, i.e., a path in $G[r]$ that alternates between edges inside and outside of M, with the first and the last edge not in M. Switching the state of all edges in an augmenting path (inserting or removing them from M) *augments* the matching, increasing its size by one.

The algorithm detects several vertex-disjoint augmenting paths at once. It computes a *layer subgraph* of $G[r]$, from which it reads off the vertex-disjoint augmenting paths. Both the construction of the layer subgraph and the search for augmenting paths are realized through a graph traversal in $G[r]$ in $O(m)$ time, where m is the number of edges. Having identified augmenting paths, the algorithm augments the matching and starts over, repeating the search until all vertices are matched or no augmenting path can be found. As shown in [68], the algorithm terminates after $O(\sqrt{n})$ rounds, yielding a running time of $O(m\sqrt{n}) = O(n^{2.5})$.

Geometry helps. The crucial observation of Efrat et al. is that for a geometric graph $G[r]$, the layer subgraph does not have to be constructed explicitly. Instead one may use a near-neighbor search data structure, denoted by $\mathcal{D}_r(S)$, which stores a point set S and a radius r. It must answer queries of the form: given a point $q \in \mathbb{R}^2$, return a point $s \in S$ such that $d(q,s) \leqslant r$. $\mathcal{D}_r(S)$ must support deletions of points in S. As the authors show, if $T(|S|)$ is an upper bound for the cost of one operation in $\mathcal{D}_r(S)$, the algorithm by Hopcroft and Karp runs in $O(n^{1.5}T(n))$ time for a graph with $2n$ vertices. For the planar case, Efrat et al. show that one can construct such a data structure (for any L_p-metric) in $O(n \log n)$ preprocessing time, with $T(n) = O(\log n)$ time per operation. Thus, the execution of the Hopcroft–Karp algorithm costs only $O(n^{1.5} \log n)$.

Naively sorting the edge weights and binary searching for the value of r

takes $O(n^2 \log n)$ time. But this running time would dominate the improved Hopcroft–Karp algorithm. In order to improve the complexity of the edge search, the authors use an approach, attributed to Chew and Kedem [42], for efficient k-th distance selection for a bi-chromatic point set under the L_∞-distance; see [60, Sec.6.2.2] for details.

With this technique, the computation of a maximum matching dominates the cost of finding the k-th largest distance, giving the runtime complexity of $O(n^{1.5} \log^2 n)$ for computing the bottleneck matching. Using further optimizations [60, Sec.5.3], they obtain a running time of $O(n^{1.5} \log n)$ for geometric graphs in $\mathbb{R}^2$ with the L_∞-metric.

It is not hard to see that the analysis carries over to the case of persistence diagrams (also mentioned in [54, p.196]). Let $G_1 = (U \sqcup V, U \times V)$ be the graph defined in Lemma 3.2. In the algorithm, $\mathcal{D}_r(S)$ is initialized with the points in V, which are subsequently removed from it. We additionally maintain a set S' of diagonal points contained in S. When the algorithm queries a near neighbor of a diagonal point of U, we return one of the diagonal points from S' in constant time, if S' is not empty. The overhead of maintaining S' is negligible. We summarize:

Theorem 3.6. *The bottleneck distance of two persistence diagrams can be computed in $O(n^{1.5} \log n)$.*

Our approach. Our implementation follows the basic structure of Efrat et al., reducing the construction of layered subgraphs to operations on a near-neighbor data-structure $\mathcal{D}_r(S)$. But instead of the rather involved data structure proposed by the authors, we use a simpler alternative: we construct a k-d tree for S. When searching for a point at most r away from a query point q, we traverse the k-d tree, pruning from the search the subtrees whose enclosing box is further away from the query than the current best candidate. When a point is removed from S, we mark it as removed in the k-d tree; in particular, we do not rebalance the tree after a removal. We also keep track of how many points remain in each subtree, so that we can prune empty subtrees from the subsequent searches. The running time per search query can be bounded by $O(\sqrt{n})$ per query, with n the number of points

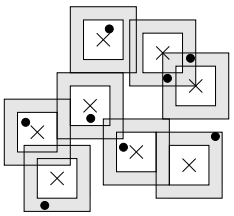


Figure 3.3: Illustration of the exact computation step: the exact bottleneck distance must be realized by a point in B (circles) in an annulus around A (crosses). The width of the annulus is determined by the approximation quality. In this example, there are 6 candidate pairs.

originally stored in the search tree. We remark that using range trees [14], the worst-case complexity could be further reduced to $O(\log n)$.

Initial tests showed that the naive approach of pre-computing and sorting all distances for the binary search dominates the running time in practice. Instead of implementing the asymptotically fast but complicated approach of Efrat et al., we compute a δ-approximation of the bottleneck distance, which we can then post-process to compute the exact answer. Let $d_{\max}$ denote the maximal L_∞-distance between a point in U and a point in V in G_1. First, we compute, in linear time, a 3-approximation of $d_{\max}$ as follows. We pick an arbitrary point in U, find its farthest point $v_0 \in V$, and find a point $u_0 \in U$ farthest from v_0. Then, $\|u_0 - v_0\|_\infty \leqslant d_{\max} \leqslant 3\|u_0 - v_0\|_\infty$ (from the triangle inequality). Setting $t = 3\|u_0 - v_0\|_\infty$, the exact bottleneck distance o must be in $[0, t]$ and we perform a binary search on $[0, t]$ until we find an interval $(a, b]$ that satisfies $(b - a) < \delta \cdot a$. We return b as the approximation. It is easy to see that $b \in [o, (1 + \delta)o)$.

At each iteration of the binary search, we reuse the maximum matching constructed before (if the true distance is below the midpoint of the current interval $(a, b]$, we remove edges whose weight is greater than $(a + b)/2$, otherwise the whole matching can be kept).

To get the exact answer, we find pairs in $U \times V$ whose distance is in the approximation interval, $(a, b]$. For such a pair (u, v), v lies in an L_∞-annulus around u with inner radius a and outer radius b. So we find for every $u \in U$ the points of V in the corresponding annulus and take the union of all such

pairs as the candidate set. In the example in Figure 3.3, points in U are drawn as crosses, points in V as circles, and there are 6 candidate pairs.

We compute the candidate pairs with similar techniques as used for range trees [14]. Specifically, we identify all pairs (u, v) whose x-coordinate difference lies in $(a, b]$. We can compute the set C_x of such pairs in $O(n \log n + |C_x|)$ time by sorting U and V by x-coordinates. For each pair (u, v) in C_x, we check in constant time whether $\|u - v\|_\infty \in (a, b]$ and remove the pair otherwise. We then repeat the same procedure using the y-coordinates. To compute the exact bottleneck distance, we perform binary search on the vector of candidate distances.

Let c denote the number of candidate pairs. The complexity of our procedure is not output-sensitive in c because $|C_x| + |C_y|$ can be larger than c — so too many pairs might be considered. Nevertheless, we expect that when using a sufficiently good initial approximation, both $|C_x| + |C_y|$ and c are small, so our method will be fast in practice.

Experiments. We compare the geometric and non-geometric bottleneck matching algorithms. We set $\delta = 0.01$ and compute the approximate bottleneck distance to the relative precision of δ, using k-d trees for the geometric version and constructing the layered graph combinatorially in the non-geometric version. Figure 3.4 shows the results for *normal* and *real* instances. We observe that the geometric version scales significantly better, and runs faster by a factor of roughly 10 for the largest displayed *normal* instance with 25000 points per diagram. We remark that the memory consumption of the geometric and non-geometric versions both scale linearly, and the geometric version is larger by a factor of roughly 4 throughout. For 25000 points, about 60MB of memory is required.

We used linear regression to fit curves of the form cn^α to the plots of Figure 3.4 (left). For the non-geometric version, the best fit appeared for $\alpha = 2.3$, roughly matching the asymptotic bound of Hopcroft–Karp. For the geometric version, we get the best fit for $\alpha = 1.4$; this shows that despite the pessimistic worst-case complexity, the algorithm tends to follow the improved geometric bound on practical instances.

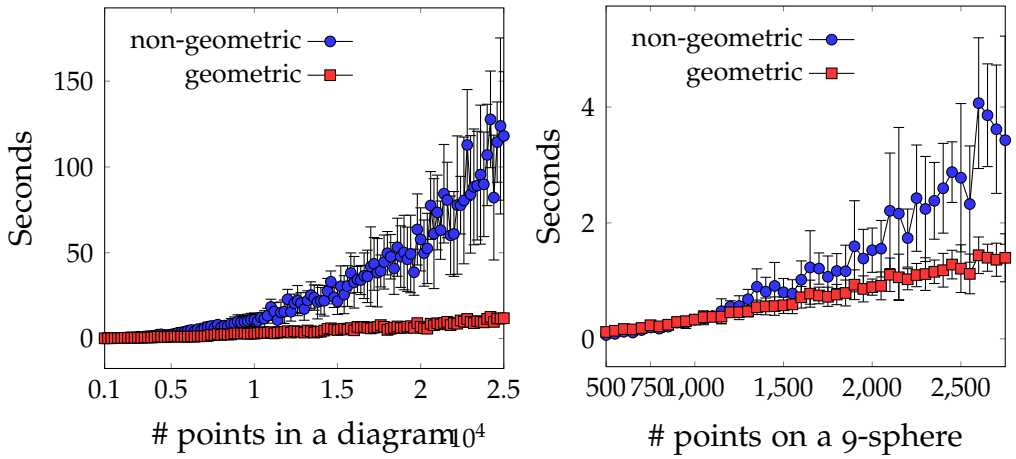


Figure 3.4: Running times of the bottleneck distance computation on *normal* data (left) and *real* data (right) for varying number of points.

The above experiment does not include the post-processing step of computing the exact bottleneck distance. We test the geometric version above that yields a 1% approximation against the variant that also computes the exact distance from the initial approximation, as explained earlier in this section. Our experiments show that the running time of the post-processing step is about half of the time needed to get the approximation. Although there is some variance in the ratio, it appears that the post-processing does not worsen the performance by more than a factor of two.

Figure 3.5 compares our exact (geometric) bottleneck algorithm with Dionysus, the only publicly available implementation for computing bottleneck distance between persistence diagrams. Dionysus simply sorts the edge distances in increasing order and performs a binary search, building the graphs $G[r]$ and calling the Edmonds matching algorithm [59] from the Boost library to check for a perfect matching in $G[r]$. Already for diagrams of 2800 points, our speed-up exceeds a factor of 400.

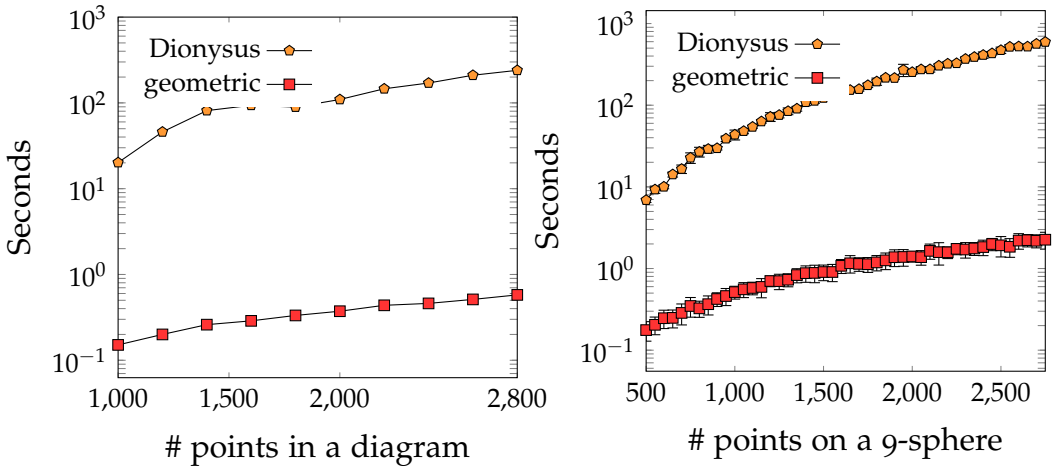


Figure 3.5: Comparison of our exact geometric bottleneck algorithm with Dionysus for *normal* (left) and *real* (right) input.

3.4 Wasserstein matchings

We now fix $q \geqslant 1$ and describe an algorithm for computing the q-Wasserstein cost of a weighted graph $(U \sqcup V, E, w)$. Recall from Proposition 3.1 that we can restrict to the 1-Wasserstein case by switching to the cost function w^q. Moreover, we assume that $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$ are finite sets, and we identify the elements with their indices.

Auction algorithm. The *auction algorithm* of Bertsekas [15] is an asymmetric approach to find a perfect matching in a weighted graph that maximizes the sum of its edge weights. One half of the bipartite graph is treated as "bidders", the second half as "objects." Initially, each object j is assigned zero price, $p_j = 0$, and each bidder i extracts a certain benefit, b_{ij}, from object j. Since we are interested in the minimum cost matching, we use the negation of the edge weight as the bidder–object benefits, that is, $b_{ij} = -w^q(i, j)$. If the edge (i, j) is not in the graph, $b_{ij} = -\infty$. The auction algorithm maintains a (partial) matching M, which is empty initially. When M becomes perfect, the algorithm stops. During the execution of the algorithm, matched bidders in M are called *assigned* (to an object), and unmatched bidders are *unassigned*.

The auction proceeds iteratively. In each iteration, one unassigned bidder i chooses an object j with the maximum value, defined as the benefit minus the current price of the object, $v_{ij} = (b_{ij} - p_j)$. Object j is assigned to the bidder; if it was assigned before, the previous owner becomes unassigned. Let Δp_{ij} denote the difference of v_{ij} and the value of the second best object for bidder i; Δp_{ij} can be zero. The price of object j increases by $\Delta p_{ij} + \varepsilon$, where ε is a small constant needed to avoid infinite loops in cases where two bidders extract the same value from two objects. Without ε, the two could keep stealing the same object from each other without increasing its price.

Our variant of the algorithm is called *Gauss–Seidel auction*: an iteration consists of only one bid, which is always satisfied. An alternative, called the *Jacobi auction*, proceeds by letting each unassigned bidder place a bid in every iteration. If several bidders want the same object, it is assigned to the bidder who offers the highest price increment, $\Delta p_{ij} + \varepsilon$. The Jacobi auction, which was used in the ALENEX paper [72], has a drawback if many objects provide the same value to many bidders. In that case, it may happen that all of these bidders bid for the same object in one iteration, and all but one of them remain unassigned. Since a Jacobi iteration is more expensive than a Gauss–Seidel iteration, this may result in worse performance. Indeed, our experiments show that switching to Gauss–Seidel auction improves the runtime by an order of magnitude.

How do we choose ε? Small values give a better approximation of the exact answer; on the other hand, the algorithm converges faster for large values of ε. Bertsekas suggests *ε-scaling* to overcome this problem: running several rounds of the auction algorithm with decreasing values of ε, using prices from the previous round, but an empty matching, as an initialization for the next round. Following the recommendation of Bertsekas and Castañon [17], we initialize ε with the maximum edge cost divided by 4 and divide ε by 5 when starting a new round.

Iterating this procedure long enough would eventually yield the exact Wasserstein distance [15]; however, the number of rounds of ε-scaling would in general be too high for many practical problems. Instead, we use a termination condition that guarantees a relative approximation of the exact value. We fix some approximation parameter $\delta \in (0, 1)$. After finishing a

round of the auction algorithm for q-Wasserstein matching for some value $\varepsilon > 0$, let $d := d_\varepsilon$ be the q-th root of the cost of the obtained matching. We stop if d satisfies

$$d^q \leqslant (1+\delta)^q (d^q - n\varepsilon), \tag{3.3}$$

and return d as the result of the algorithm. We summarize the auction in Algorithm 1.

Lemma 3.7. *The return value d of the algorithm satisfies*

$$d \in [o, (1+\delta)o),$$

where o denotes the exact q-Wasserstein distance.

Proof. Because we raise all edge costs to the q-th power, the matching minimizing the sum of the edge costs has a cost of o^q. Let d^q be the cost of the matching computed by the auction algorithm, after the last round of ε-scaling, for a fixed ε. By the properties of the auction algorithm ([16], Proposition 1), it holds (after every round) that

$$o^q \leqslant d^q \leqslant o^q + n\varepsilon.$$

Taking the q-th root yields the first inequality immediately. For the second inequality, note that

$$(1+\delta)^q o^q \geqslant (1+\delta)^q (d^q - n\varepsilon) \geqslant d^q,$$

where the last inequality follows from the termination condition of the algorithm. Taking the q-th root on both sides yields the result. □

Bidding. The computational crux of the algorithm is for a bidder to determine the object of maximum value and the price increase. The brute-force approach is for each bidder to do an exhaustive search over all objects. Doing so requires linear running time per iteration. But let us consider what the search actually entails. Bidder i must find the two objects with highest and second-highest v_{ij} values. Recall $v_{ij} = b_{ij} - p_j = -w^q(i,j) - p_j$, and maximizing this quantity for a fixed i is equivalent to minimizing $w^q(i,j) + p_j$.

Algorithm 1 AUCTION ALGORITHM

function AUCTION(X, Y, q, δ)
▷ Input: two persistence diagrams X, Y with $|X|, |Y| \leqslant n$, $q \geqslant 1$, $\delta > 0$ (maximal relative error)
▷ Output: δ-approximate q-Wasserstein distance $W_q(X, Y)$
  Initialize $d \leftarrow 0$ and $\varepsilon \leftarrow \frac{5}{4} \cdot$ (max. edge length)
  while $d^q > (1+\delta)^q (d^q - n\varepsilon)$ **do**
    $\varepsilon \leftarrow \varepsilon/5$
    Let M be an empty matching
    while there exists some unassigned bidder i **do**
      Find the best object j with value v_{ij} and the second best object k with value v_{ik} for i
      Assign j to i in M and increase the price of j by $(v_{ij} - v_{ik}) + \varepsilon$
    $d \leftarrow q$-th root of the cost of the (perfect) matching M
  return d

The first way to quickly find these objects uses lazy heaps. Each bidder keeps all the objects in a heap, ordered by their value. We also maintain a list of all the price changes (for any object), as well as a record for each bidder of the last time its heap was updated. Before making a choice, a bidder updates the values of all the objects in its heap that changed prices since the last time the heap was updated. The bidder then selects the two objects with the maximum value. We note that this approach uses quadratic space, since each bidder keeps a record of each object.

The second way to accelerate the search for the best object uses geometry and requires only linear space. Initially, when all the prices are zero, we can find the two best objects by performing the proximity search in a k-d tree. But we need to augment the k-d tree to take increasing prices into account. We do so by storing the price of each point as its weight in the k-d tree. At each internal node of the tree we record the minimum weight of any node in its subtree. When searching, we prune subtrees if the q-th power of the distance from the query point to the box containing all of the subtree's points, plus the minimum weight in the subtree, exceeds the current second best candidate.

Once a bidder selects the best object, it increases its price. We adjust the

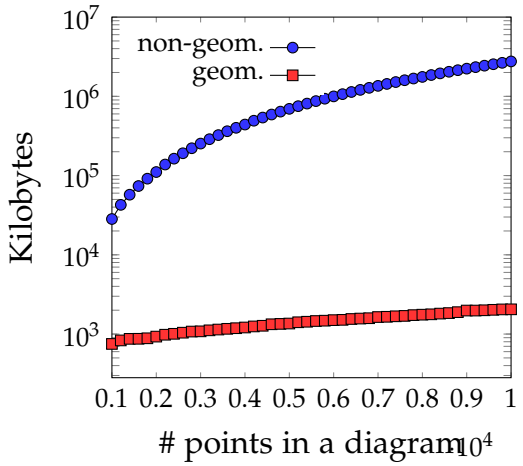


Figure 3.6: Comparison of memory consumption of geometric and non-geometric versions of auction algorithm on *normal* instances.

subtree weights in the k-d tree by increasing the chosen object's weight and updating the weights on the path to the root. If the minimum weight does not change at some node on the path, we interrupt the traversal.

The case of persistence diagrams requires special care. We can distinguish between *diagonal* and *off-diagonal* bidders and objects. Diagonal bidders should bid for only one off-diagonal object, according to Lemma 3.3. Since the distance between diagonal points is 0, the value of a diagonal object j for a diagonal bidder i is just the opposite of its price, $v_{i,j} = -p_j$, and we keep all diagonal objects in a heap ordered by the price. When a diagonal bidder needs to find the best two objects, it selects the top two elements of the heap and compares them with the only off-diagonal object to which it can be assigned.

On the other hand, off-diagonal bidders can bid for every off-diagonal object and only for one diagonal object (its projection). We use one global k-d tree to get the best two off-diagonal objects and then compare their values for the bidder with the value of bidder's projection, so only off-diagonal objects are stored in the k-d tree.

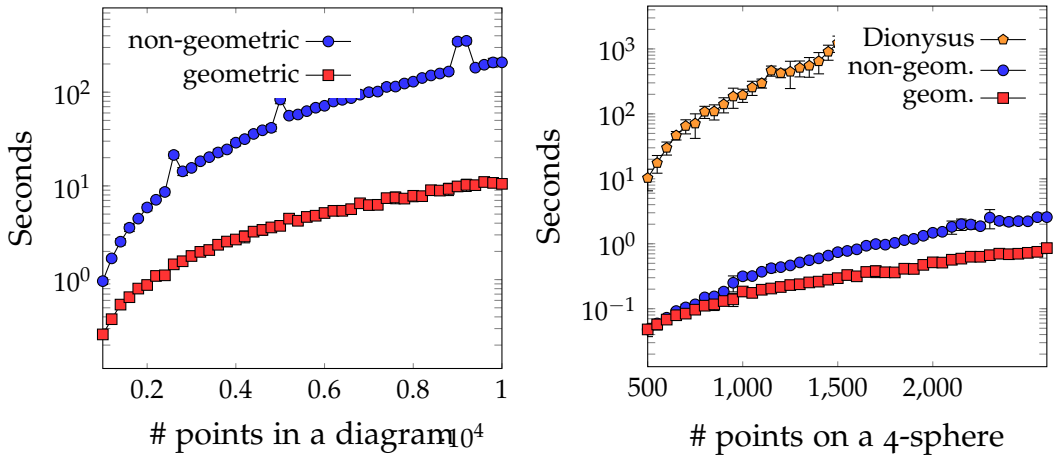


Figure 3.7: Comparison of non-geometric and geometric variants of the auction algorithm on *normal* (left) and *real* (right) input, also with Dionysus on the *real* input.

Experiments. Figure 3.7 illustrates the running times of the auction algorithm on the *normal* data, using lazy heaps and k-d trees. In both cases, we compute a relative 0.01-approximation. The advantage of using geometry is evident: the algorithm is faster by roughly a factor of 4 for diagrams with 1000 points, and the factor becomes close to 20 for diagrams with 10000 points. We used linear regression to empirically estimate the complexity, and the geometric algorithm runs in $O(n^{1.6})$, while for the non-geometric algorithm the estimated complexity is super-quadratic, $O(n^{2.3})$. The non-geometric version only shows competitive running times because of the described optimization with lazy heaps. This results in a severe increase in memory consumption, as displayed in Figure 3.6.

Again, we compare our geometric approach with Dionysus, which uses John Weaver's implementation[3] of the Hungarian algorithm [85]. Figure 3.7 (right) shows the results for *real* instances. The speed-up of our approach increases from a factor of 50 for small instances to a factor of about 400 for larger instances. For the *normal* data sets, the speed-up already exceeds a factor of 1000 for diagrams of 1000 points; we therefore omit a plot.

We emphasize that our test is slightly unfair, as it compares the exact

[3] `http://saebyn.info/2007/05/22/munkres-code-v2/`

algorithm from Dionysus with the 0.01-approximation provided by our implementation. While such an approximation suffices for many applications in topological data analysis, the question remains how much overhead would be caused by an exact version of the auction algorithm. A naive approach to get the exact result is to rescale the input to integer coordinates and to choose ε such that the approximation error is smaller than 1. We plan to investigate different possibilities to compute the exact distance more efficiently.

3.5 Wasserstein matchings for repeated points

For a weighted, complete, bipartite graph $G = (U \sqcup V, U \times V, w)$, we call two vertices $u_1, u_2 \in U$ *identical* if for all $v \in V$, $w(u_1, v) = w(u_2, v)$. A pair of identical vertices in V is defined symmetrically. If G is a geometric graph, two points with coinciding locations are identical. In the context of persistence diagrams, this situation is common in applications, where the range of possible scales on which features appear and disappear is often discretized. The discretization places all points of the persistence diagram on a finite grid. For a fixed discretization of a fixed range, more and more identical points appear as the data size grows. This raises the question whether diagrams with many identical points can be handled more efficiently.

Auction with integer masses. We use a variant of the auction algorithm [18], which we explain next. The input consists of two sets U and V of *multi-points*, each given by its coordinates and integer multiplicity $m \geqslant 1$; a multi-point represents m identical points at the given location. For brevity, we refer to the multiplicity as *mass*. The total mass of both sets is the same. In analogy to the auction algorithm from Section 3.4, we refer to the elements of the respective sets as *multi-bidders* and *multi-objects*. The elements of a multi-object are not, in general, assigned to the same multi-bidder; their prices can also differ. However, if two elements of a multi-object are assigned to one multi-bidder, the algorithm guarantees that their prices are equal. The algorithm decomposes a multi-object into *slices*, where each slice represents a fraction

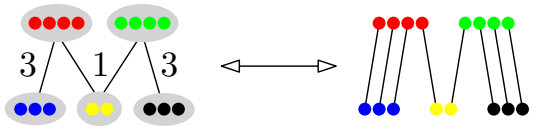


Figure 3.8: Correspondence between assignments and matchings. On the left-hand side there are two multi-bidders, each of mass 4, and 4 slices with masses 3, 3, 1, 1. A corresponding perfect matching is on the right-hand side.

of the multi-object that is currently not distinguished by the algorithm. Formally, a slice is a four-tuple $(j, m_{i,j}, p_{i,j}, i)$ identifying the multi-object j it belongs to, the mass of the slice $m_{i,j}$, its price $p_{i,j}$, and the multi-bidder i that it is currently assigned. The decomposition of multi-objects into slices defines an *assignment*, which can be interpreted as a matching M in the original graph (see Figure 3.8): A slice $(j, m_{i,j}, p_{i,j}, i)$ corresponds to $m_{i,j}$ edges in M from $m_{i,j}$ elements of the multi-bidder i to $m_{i,j}$ elements of the multi-object j (hereby interpreting multi-bidders and multi-objects as sets of identical bidders/objects). Unassigned slices correspond to unmatched vertices. We call an assignment *perfect* if the induced matching is perfect, and the *cost* of the assignment is the cost of the corresponding matching.

The auction with integer masses is a procedure converging to an assignment with minimal cost. It uses the same high-level structure as the auction described in Section 3.4, which we will refer to as the *standard auction*. It employs ε-scaling with the same choices of parameters. One round of ε-scaling maintains an assignment and runs until the assignment is perfect, that is, all multi-bidders are fully assigned to multi-objects. Every round proceeds in iterations. In each iteration, one multi-bidder with unassigned mass is selected at random. It acquires enough slices (possibly taking them away from other multi-bidders) to assign all its missing mass and increases the prices of these slices.

Specifically, an iteration proceeds as follows. We fix a multi-bidder with some unassigned mass $u \geqslant 1$, and let $s_1, \dots, s_t$ be the slices assigned to it. Conceptually, the algorithm takes all possible slices except for $s_1, \dots, s_t$ and sorts them by their value to the multi-bidder in decreasing order. We denote the sorted slices by $s_{t+1}, \dots, s_N$; let v_i denote the value of s_i to the multi-bidder.

The multi-bidder takes the first k slices $s_{t+1}, \ldots, s_{t+k}$ such that their total mass m is at least u. If $m > u$, we split the "leftover" slice from s_{t+k} whose mass is $m - u$ and whose price and owner remain unchanged; we denote this newly created slice as $\tilde{s}_{t+k}$. Now, the total mass of the slices $s_{t+1}, \ldots, s_{t+k}$ is exactly u, and we assign them to the multi-bidder.

Next, we increase the prices of all $t + k$ slices assigned to the multi-bidder. Let s_l with $l \geqslant t + k$ be a slice determined as follows: if the slices $s_1, \ldots, s_{t+k}$ belong to at least two different multi-objects, s_l is the slice containing the $(m + 1)$-st unit of mass, that is, s_l is set to s_{t+k+1} if we did not split the leftover slice, and to $\tilde{s}_{t+k}$ otherwise. If all the $t + k$ slices are of a single multi-object, then s_l is defined to be the first slice among $s_{t+k+1}, \ldots, s_N$ that belongs to a different multi-object. Let v_l be the value of s_l to the multi-bidder. We increase the prices of the slices $\{s_i\}_{1 \leqslant i \leqslant t+k}$ by $v_i - v_l + \varepsilon$ to make them as valuable to the multi-bidder as the slice s_l, up to ε.

The original paper that presents this approach [18] describes the Jacobi version of the algorithm, i.e., all bidders with unassigned mass submit bids in one iteration, and the mass goes to the bidder who offered the highest bid. The above description is the Gauss–Seidel variant of the same algorithm, and it is straightforward to verify that the same proof of correctness works for it, too. From the discussion in [18], it follows that one can use the same formula as for the standard auction to estimate the relative error of the matching obtained after each round of ε-scaling. Therefore, we can use the same termination condition as in (3.3) and the proof of Lemma 3.7 carries over. We refer to [18] for further details.

Diagonal points. Let X_0 and Y_0 be the of the off-diagonal mult-points of two persistence diagrams. Recall that for the computation of the q-Wasserstein distance, we introduce the projection sets Y_0' and X_0' (also as a set of multi-points with masses inherited from their pre-images) and set $X := X_0 \cup Y_0'$ and $Y := Y_0 \cup X_0'$, which are sets of multi-points with equal total mass. We can run the auction with integer masses, using the cost function c^q, with c as in (3.1), and return the q-th root of the obtained cost as our result. However, we get a major improvement from using the modified cost function $\tilde{c}$, defined in (3.2). The modified function decreases the costs

of all skew edges; accordingly, $\tilde{c}^q$ treats all points in X_0' as identical and all points in Y_0' as identical.

In terms of the auction with integer masses, this means that we only need one additional multi-bidder (with large mass) to represent all projections of multi-objects to the diagonal, and vice versa. Specifically, writing $X_0 = \{x_1, \dots, x_k\}$ for the off-diagonal multi-bidders, let m_X denote their total mass. Let $Y_0 = \{y_1, \dots, y_\ell\}$ denote the off-diagonal multi-objects with total mass m_Y. We introduce one additional multi-bidder $Y_0' := \{x_{k+1}\}$ (representing all projections of multi-objects), with mass m_Y, and one additional multi-object $X_0' := \{y_{\ell+1}\}$ with mass m_X. The bidder–object benefits are set up according to (3.2) (recall that x_i' denotes the projection of x_i onto the diagonal):

$$b_{i,j} = \begin{cases} -\|x_i - y_j\|_\infty^q, & i \leqslant k \text{ and } j \leqslant \ell \\ -\|x_i - x_i'\|_\infty^q, & i \leqslant k \text{ and } j = \ell + 1 \\ -\|y_j - y_j'\|_\infty^q, & i = k + 1 \text{ and } j \leqslant \ell \\ 0, & i = k + 1 \text{ and } j = \ell + 1 \end{cases}$$

Implementation. We implemented a geometric version of the auction with integer masses, where the best slices of the off-diagonal multi-objects are determined using one global k-d tree, similar to Section 3.4. Here, each leaf of the k-d tree represents a multi-object, and its weight corresponds to the price of its cheapest slice. For a fixed off-diagonal multi-bidder, we can compute an upper bound on the value of all multi-objects stored in a subtree of the k-d tree. During a search, we maintain a candidate set of slices whose total mass exceeds the unassigned mass of the multi-bidder, and we can prune a subtree if that upper bound is below the value of the worst candidate. The weights in the k-d tree are updated as in Section 3.4. The additional information required to compute the price increases are gathered by similar techniques; we omit the details. We did not implement a non-geometric version using lazy heaps because it would suffer from the same quadratic space complexity as in the standard auction.

Again we need to deal with the diagonal multi-object and multi-bidder separately. We maintain a heap with the slices of the diagonal multi-object

sorted by the price and a heap with the slices of all multi-objects (including the diagonal one) sorted by their value for the diagonal bidder. The diagonal bidder finds the best slices by simply traversing the latter heap. An off-diagonal bidder first uses the k-d tree to find the best slices of off-diagonal objects. Then it starts traversing the heap with slices of the diagonal object, replacing the off-diagonal slices with the diagonal ones as long as the diagonal slices offer better values. When the value of the next diagonal slice in the heap is below the minimal value of the currently accumulated slices, we stop traversing the heap with diagonal slices. When slice prices are increased, we immediately update the heaps.

Experiments. As input, we turn the aforementioned instances of *normal* type into diagrams with integer masses. For each point of the original diagram, we assign mass m, drawn uniformly from the range $[\lceil k/2 \rceil, \lfloor 3k/2 \rfloor]$, so that the average mass of a point is k. In our experiments, we compare the standard auction and the auction with integer masses for $k = 1, 10, 50, 100$.

We generated *normal* instances with 1,000 to 10,000 points, in increments of $1,000$, with 10 instances per size. Figure 3.9 shows the average running times. There is an overhead for mass 1 (a factor of roughly 4.5 in the figure). This ratio is not constant: the overhead becomes larger when the number of points grows. We also observe that it depends on the parameters of the distribution from which the points were drawn. For average mass 10, the auction with masses is comparable to the standard auction. For higher masses, 50 and 100, the advantage of the former is evident.

There is no clear dependence between the running time and the average mass. We took 4 instances with 10000 points each and tried larger average masses (with the same $[\lceil k/2 \rceil, \lfloor 3k/2 \rfloor]$ distribution). Figure 3.10 illustrates the result. We can see that the running time does not increase much when the average mass increases, and may even decrease. That seems to depend very much on the particular instance and the distribution of masses inside it.

The memory consumption of auction with integer masses usually scales linearly with the number of points (for fixed average mass). In principle, the memory size can grow proportional to the total mass of the point sets when

all slices shrink to size one, but such intensive slicing did not appear in our examples.

3.6 Parallelization of Wasserstein distance computation

One of the reasons for choosing the auction algorithm was that it is easy to parallelize. We had a very concrete case in mind: computing Wasserstein distance (for $q = 2$ or $q = 3$) between diagrams with approximately 10^6 points in a massively parallel setting (using $10^4 - -10^5$ cores). Shared memory parallelism is not so interesting for this example, because the number of cores on a typical machine is rarely larger than 16, in most cases being 8. Even if we had perfect scaling, acceleration by a factor of 16 would not be enough for processing diagrams of this size. In this short section, which is not published in [72], we discuss a problem that we encountered (and could not solve) trying to make a parallel version efficient.

A natural choice for parallelization is the Jacobi variant of the algorithm, in which during the bidding phase each unassigned bidder must submit a bid, and then each item that received a bid goes to the bidder who offered more than others. The expensive part of this process is searching for the most attractive item and exactly this process can be parallelized, since in the bidding phase the bidders do not interact with each other at all. However, we immediately discovered that most of the bidders are assigned in the first rounds, and in a long sequence of the following rounds, only a small fraction of bidders has to submit bids. Figure 3.11 shows how the number of unassigned bidders decreases as the auction algorithm is running; the data is for a random sample of cardinality 50,000 from a real diagram (the total number of bidders is 100,000, because we add diagonal projections). Since we are interested in massively parallel setting, these long tails with small number of unassigned bidders make such a trivial parallel variant practically useless.

We can try the following heuristics. If there are only a few unassigned bidders, then their contribution to the total cost of the final matching is

probably negligible. If we can find a way to guarantee lower and upper bounds for the Wasserstein cost after each round of the algorithm, when some bidders are unassigned, and not only at the end, when we have a perfect matching, we could stop the algorithm earlier.

For a point $p = (p_x, p_y)$ of a persistence diagram, let $pers(p)$ denote its *persistence*, i.e., $\mathrm{pers}(p) = (p_y - p_x)/2$. Thus, $pers(p)$ is the cost of matching p with its projection on the diagonal.

Lemma 3.8. *Let $\tilde{M}$ be a partial matching computed by the auction algorithm at the end of some round, and let γ be defined by the equation*

$$\gamma^q = \Sigma' pers^q(b) + \sigma'' pers^q(i),$$

where $\sum'$ is taken over all unassigned bidders b, and $\sum''$ is taken over all unassigned items i. If we return $\sqrt[q]{c(\tilde{M})}$ as an approximation of the Wasserstein distance, then the relative error is at most

$$\frac{\gamma + \sqrt[q]{c(\tilde{M})} - \sqrt[q]{c(\tilde{M}) - n\varepsilon}}{\sqrt[q]{c(\tilde{M}) - n\varepsilon}},$$

where $c(\tilde{M})$ is the cost of $\tilde{M}$.

Proof. We need to show that $c(\tilde{M}) + \gamma$ is an upper bound for the optimal cost, and $c(\tilde{M}) - n\varepsilon$ is a lower bound for the optimal cost. Assuming these statements, the lemma follows from a simple calculation similar to the one made in the proof of Lemma 3.7.

The upper bound part becomes obvious, if we use the definition of the Wasserstein distance that takes infimum over all bijections between all points of the persistence diagrams. The partial matching $\tilde{M}$ defines a bijection between the assigned bidders and items, and all the rest is mapped to the diagonal; the cost of the resulting bijection is exactly the sum of the cost of $\tilde{M}$ and γ, and this cost cannot be less than the cost of an optimal bijection.

The lower bound part easily follows from the properties of the auction algorithm. Recall that for each assigned bidder the ε-complementary slackness conditions are satisfied. This means that the cost of the matching $\tilde{M}$ is by

at most $n\varepsilon$ greater than the cost of the maximal (non-perfect) matching of the bipartite graph obtained by deleting the bidders that are currently unassigned. Deletion of some of the bidders can only decrease the optimal cost, since all items are kept. □

Thus we can maintain the number γ and stop the algorithm before reaching a perfect matching, using the lemma to guarantee that the cost of the partial matching is close enough to the true optimal cost. Note that updating γ is easily done in constant time, along with updating the matching. We tested this optimization on some relatively small examples. First, we determined the value of ε that will yield the desired relative error, and then we ran the normal auction algorithm and the variant that maintains γ. As expected, this early termination really helped to eliminate the long tail of non-parallelizable rounds. However, if we start with ε small enough to guarantee some reasonable relative error, say, 0.1, then the number of rounds increases (compared to the ε-scaling variant) by a factor that cannot be compensated by parallelization. We also tried using this lemma as a heuristics, stopping the auction early at each phase of ε-scaling. The outcome was even worse: the algorithm does not terminate in reasonable time at all. The reason for this behavior is that by stopping earlier phases of ε-scaling before a perfect matching is computed, we introduce large errors to the price values. While the long tail of rounds with a small number of unassigned bidders changes the cost of the current matching only slightly, it performs another very important task: computing better approximation of the prices for the next phase, and the quality of this approximation is crucial.

3.7 Conclusion

We have demonstrated that geometry helps to compute bottleneck and Wasserstein distances. Our approach leads to a faster computation of distances between persistence diagrams. Therefore, we expect our software to have an immediate impact on the computational pipeline of topological data analysis.

For bottleneck matchings, an interesting question would be how our k-d tree implementation compares in practice with the (theoretically) more time efficient, but more space demanding alternative of range trees, and with other point location data structures.

For Wasserstein matchings, we plan to further improve our implementation of the auction algorithm. Simple heuristics can also improve special cases. For example, if X and Y are persistence diagrams and $S \subset X \cap Y$ is the set of common off-diagonal points, it holds for $q = 1$ that $W_q(X, Y) = W_q(X \setminus S, Y \setminus S)$, as one shows very easily. This property allows to remove common points in the diagram before applying the auction algorithm. We also wonder how the auction approach compares with the various alternatives proposed in [31], and for which of these approaches can geometry help compute the Wasserstein distance efficiently, either exactly or approximately.

A natural approximation scheme for computing the Wasserstein distance for very large instances consists of placing a finite grid over $\mathbb{R}^2$ and "snapping" points to their closest grid vertex. The result is an instance with a potentially high multiplicity in each grid vertex. The problem with this approach is the approximation error introduced by the discretization step. A crude error bound is the total number of points in both diagrams multiplied by the diameter of the grid cells. An interesting question is to evaluate more refined discretization schemes with respect to their practical performance.

Acknowledgements

We thank Sergio Cabello for pointing out that the worst-case complexity of k-d trees and range trees remains valid under deletions of points, and for further valuable remarks on an earlier draft.

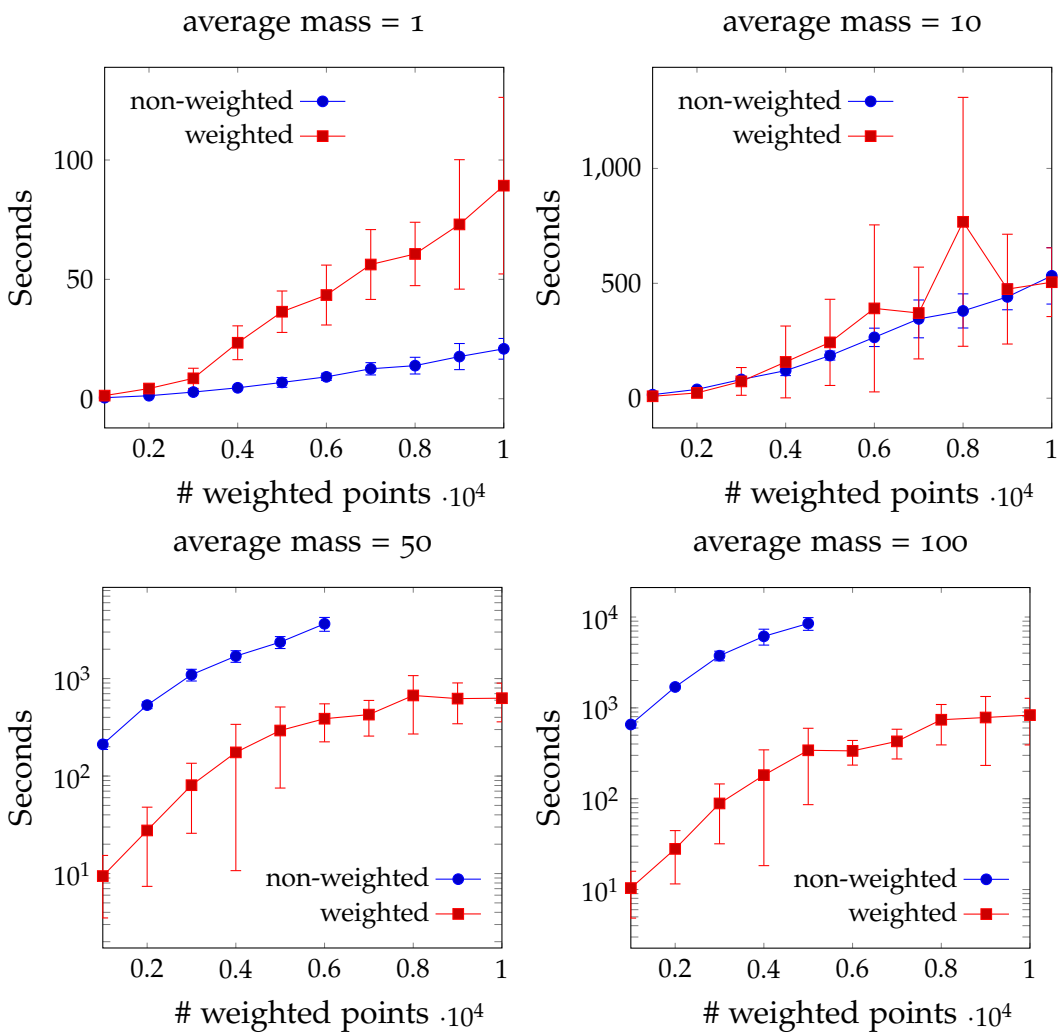


Figure 3.9: Comparison of non-weighted and weighted variants of the auction algorithm on *normal* data for mass 1 (upper-left) and average masses 10 (upper-right), 50 (lower-left) and 100 (lower-right).

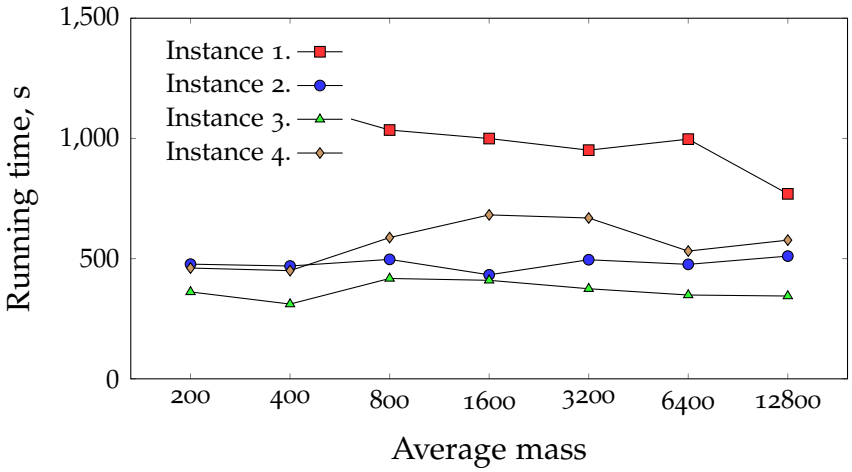


Figure 3.10: Dependence of the running time from the average mass for four particular instances of size 10000. Note the exponential scale on the x-axis.

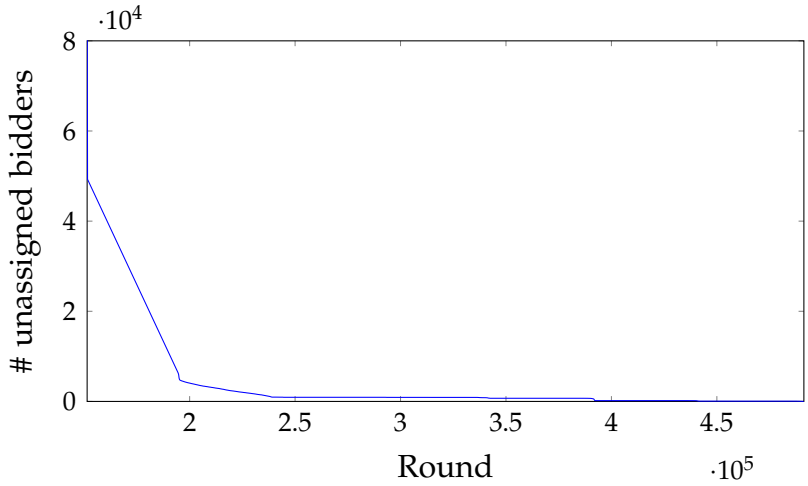


Figure 3.11: # unassigned bidders in the 3rd phase of ε-scaling (Jacobi variant of the auction algorithm). Data is for two diagrams with $50,000$ off-diagonal points in each. Numeration of rounds (x-axis) started from the 1st phase.

4 Efficient Computation of Matching Distance

This chapter is based on a manuscript authored by Michael Kerber and me.

4.1 Introduction

As mentioned in Chapter 1, approximation of the interleaving distance is NP-hard. In this chapter, we work with the matching distance, which provides a computationally tractable lower bound on D_I. The starting point for our approach is the paper [20] by Biasotti et al.

Let us summarize their algorithm. We parameterize the space of all lines of interest as a bounded rectangle $R \subset \mathbb{R}^2$. For a point p in the rectangle, let $f(p)$ denote the bottleneck distance of the two persistence diagrams obtained from the restrictions of the bi-filtrations to the line parameterized by p. The major ingredient of the algorithm is a *variation bound* which tells how much can f vary when each parameter is perturbed by a fixed amount. For any subrectangle $S \subseteq R$ with center c, knowing $f(c)$ and the variation bound yields an upper bound of f within S. We then obtain an ε-approximation with a simple branch-and-bound scheme, subdividing R with a quad-tree in BFS order and stopping the subdivision of a rectangle when its upper bound is sufficiently small.

Our contributions are the following:

1. We simplify the approximation algorithm of [20]. In particular, we eliminate all trigonometric functions — our algorithm needs to compute only polynomial expressions.
2. We provide a simple yet crucial algorithmic improvement: instead of using the *global* variation bound for all rectangles of the subdivision, we derive adaptive *local* variation bounds for each rectangle individually. This results in much smaller upper bounds and avoids many subdivisions in the approximation algorithm.
3. We experimentally compare our version of the global bound with the usage of the adaptive bounds. We show that the speedup factor of the sharpest adaptive bound is typically between 3 and 5, depending on the input bi-filtrations.
4. We plan to release our code as part of the Hera library.

4.2 Preliminaries

The general definition of the matching distance was given in definition 2.34. However, in this chapter we will only consider persistence modules coming from 1-critical bi-filtrations. We will write a bi-filtration of a complex K as (K, φ), where the function $\varphi\colon K \to \mathbb{R}^2$ provides critical values of all simplices. If $\varphi(\sigma) = (p_x, p_y)$, then the simplex σ is present in the bi-filtration at all points of the upper-right quadrant of (p_x, p_y).

Slices. For all concepts in this paragraph, see Figure 4.1 for an illustration. We will refer to a non-vertical line L with positive slope as a *slice*. For every slice, we distinguish a point $\mathcal{O}$, called the *origin* of the slice. We let $\mathcal{L}$ denote the set of all slices. Since the slope is positive, for any two distinct points p, q on L either $p \preceq q$ or $q \preceq p$ holds. Hence, $\preceq$ becomes a total order along L.

Recall the definition of the weight $w(L)$. Let γ denote the angle between the slice L and x-axis. We call L *flat* if $\gamma \leqslant \frac{\pi}{4}$ (i.e., if its slope is at most 1) and *steep* if $\gamma \geqslant \frac{\pi}{4}$. Then we set

$$w(L) := \begin{cases} \sin\gamma & \text{if } L \text{ is flat} \\ \cos\gamma & \text{if } L \text{ is steep.} \end{cases}$$

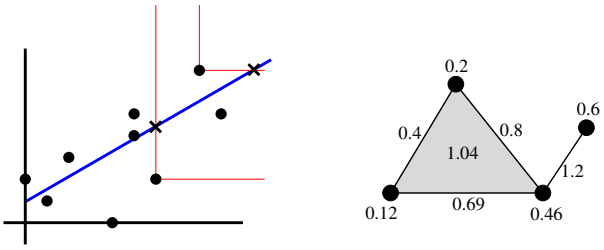


Figure 4.1: Left: The slice parameterized by $(\frac{\pi}{6}, 0.1)$. For two critical values of the bi-filtration from above, we illustrate the construction of the point q (displayed by a cross shape). The push of the critical value is then simply the Euclidean distance to the point $(0, 0.1)$, which is the origin of the slice. Right: The non-weighted restriction on the slice. Each simplex gets its push as critical value (values rounded to two digits).

Given $p \in \mathbb{R}^2$, let q be the minimal point on L (with respect to $\leqslant$) such that $p \leqslant q$. Geometrically, q is the intersection of L with the boundary of the upper-right quadrant of p, or equivalently, the horizontally-rightwards projection of p to L if p lies above L, or the vertically-upwards projection of p to L if p lies below L. Since q lies on L, q can be uniquely written as

$$\mathcal{O} + \lambda_p \begin{pmatrix} \cos\gamma \\ \sin\gamma \end{pmatrix}$$

where γ is the angle between L and x-axis, and $\lambda_p \in \mathbb{R}$. We say that λ_p is the *push* of p to L, which can be formally written as a function $\text{push} : \mathbb{R}^2 \times \mathcal{L} \to \mathbb{R}$. Geometrically, the push is simply the (signed) distance from the point q to the origin of the slice. For a bi-filtration $F = (K, \varphi)$, the composition $\text{push}(\cdot, L) \circ \varphi$ is a function $K \to \mathbb{R}$. This function yields a mono-filtration of K, which we call the *non-weighted restriction* of F onto L. The non-weighted restriction will be denoted $restr(F, L)$. See Figure 4.1, right for an example.

Recall that the matching distance between two bi-filtrations $F^1 = (K^1, \varphi^1)$ and $F^2 = (K^2, \varphi^2)$ is

$$\mathrm{d_M}(F^1, F^2) := \sup_{L \in \mathcal{L}} w(L) \cdot W_\infty(restr(F^1, L), restr(F^2, L)),$$

where W_∞ is actually the bottleneck distance between the persistence modules obtained by applying a homology functor $H_n(\cdot, \mathbb{F})$ to the non-weighted restrictions.

Since W_∞ is shift-invariant, we can assume that all critical values of $F^{1,2}$ are in the positive (upper-right) quadrant.

Let us now define the *weighted push* of a point p to a slice L as

$$\mathrm{wpush}(p, L) = w(L)\,\mathrm{push}(p, L),$$

and let F_L denote the mono-filtration induced by $\sigma \mapsto \mathrm{wpush}(\varphi(\sigma), L)$. We call F_L a *weighted restriction* of F onto L. Note that F_L equals $restr(F, L)$ except that all critical values are scaled by the factor $w(L)$. Using homogeneity of W_∞, we see that we can define the matching distance as

$$\mathrm{d_M}(F^1, F^2) := \sup_{L \in \mathcal{L}} W_\infty(F^1_L, F^2_L). \tag{4.1}$$

We will use this equivalent definition of the matching distance in the remaining part of the chapter, and 'restriction' will always mean 'weighted restriction'.

4.3 The approximation algorithm

The idea of the approximation algorithm for $\mathrm{d_M}$ is to sample the set of slices through a finite sample, and choose the maximal bottleneck distance between the (weighted) restriction encountered as the approximation value. In order to execute this plan, we need to parameterize the space of slices and to compute the restriction of a parameterized slice efficiently.

Slice Parameterization. Every slice has a unique point where the line enters the positive quadrant of $\mathbb{R}^2$, which is either its intersection with the positive x-axis, the positive y-axis, or the point $(0, 0)$. From now on, we always use this point as the origin of the slice.

We call a slice an *x-slice* if its origin lies on the positive x-axis, and call it a *y-slice* if its origin lies on the positive y-axis (slices through the origin are both x- and y-slices). Recall also that a slice is flat if its slope is less than 1, and steep if it is larger than 1. Thus, a slice belongs to one of the four

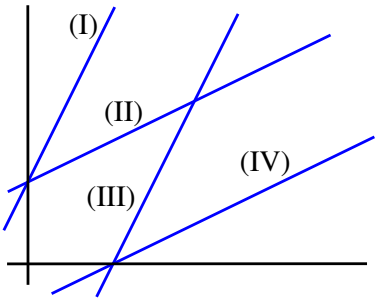


Figure 4.2: A steep y-slice (I), a flat y-slice (II), a steep x-slice(III) and a flat x-slice. The slopes are 2 for the steep and $\frac{1}{2}$ for the flat slices, and the origin is at $(0,2)$ for the y-slices and at $(2,0)$ for the x-slices. Consequently, all four slices are parameterized by $(\frac{1}{2},2)$.

types: flat x-slices, flat y-slices, steep x-slices and steep y-slices. Every slice is represented as a point $(\lambda,\mu) \in (0,1] \times [0,\infty)$ where the interpretation of the parameters depends on the type of the slice as follows: Recall that γ is the angle of the slice with the x-axis. Then

$$\lambda = \begin{cases} \tan(\gamma), \text{ if } L \text{ is flat} \\ \cot(\gamma), \text{ if } L \text{ is steep} \end{cases},$$

in other words, λ is the slope of the line in the flat case, and the inverse of the slope in the steep case. With $\mathcal{O} = (\mathcal{O}_x, \mathcal{O}_y)$ the origin of L,

$$\mu = \begin{cases} \mathcal{O}_x & \text{if } L \text{ is } x\text{-slice} \\ \mathcal{O}_y & \text{if } L \text{ is } y\text{-slice} \end{cases}.$$

Note that the same pair of parameters can parameterize different slices depending on the type. Figure 4.2 illustrates this.

Weighted pushes. We next show a simple formula for the value of $\text{wpush}(p,L)$ depending on the type of the slice.

Lemma 4.1. *With the chosen parameterization and choice of origin on L,* $\text{wpush}(p,L)$ *is computed according to the formulas given in Table 4.1 and Table 4.2.*

y-slices	L is flat	L is steep
p above L	$p_y - \mu$	$\lambda(p_y - \mu)$
p below L	λp_x	p_x

Table 4.1: Formulas for weighted push of (p_x, p_y) onto a y-slice $L = (\lambda, \mu)$.

Proof. The proof of the lemma is a series of elementary calculations. Let us consider, for example, the case of a flat y-slice. In this case the slice $L = (\lambda, \mu)$ is given by

$$\left\{ \begin{pmatrix} 0 \\ \mu \end{pmatrix} + \rho \begin{pmatrix} \cos\gamma \\ \sin\gamma \end{pmatrix} \mid \rho \in \mathbb{R} \right\},$$

If $p = (p_x, p_y)$ is above L, we consider the point $q = (q_x, q_y)$ which is the intersection of L and the line $y = p_y$. Obviously, $q_y = p_y$, and, since q lies on L, the second coordinate yields the equation

$$\rho = \frac{q_y - \mu}{\sin\gamma} = \frac{p_y - \mu}{\sin\gamma}.$$

By definition, ρ is the push of p to L and since L is flat, we have that $w(L) = \sin\gamma$.

If p is below L, we have, by the first coordinate,

$$\rho = \frac{q_x}{\cos\gamma} = \frac{p_x}{\cos\gamma},$$

and multiplying with the weight yields

$$\mathrm{wpush}(p, L) = \sin\gamma \frac{p_x}{\cos\gamma} = p_x \tan\gamma = p_x\lambda$$

The other cases are proved analogously. □

All 8 expressions in Table 4.1 and 4.2 involve only addition and multiplication without trigonometric functions. Hence we can extend them continuously to case $\lambda = 0$, which corresponds to horizontal lines (in the flat case) or vertical lines (in the steep case). With this interpretation, we can

x-slices	L is flat	L is steep
p above L	p_y	λp_y
p below L	$\lambda(p_x - \mu)$	$p_x - \mu$

Table 4.2: Formulas for weighted push of (p_x, p_y) onto an x-slice $L = (\lambda, \mu)$.

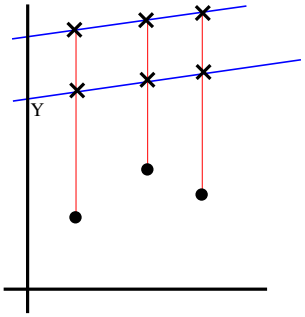


Figure 4.3: Illustration for the fact that slices with larger value of μ can be ignored.

extend $\mathcal{L}$ to a set $\bar{\mathcal{L}}$ in (4.1), containing these limit cases, without changing the supremum.

Next, we observe that we can restrict our attention to a bounded range of μ-parameters. For that, let X denote the maximal x-coordinate and Y be the maximal y-coordinate among all critical values of F^1 and of F^2. For a y-slice (steep or flat) $L = (\lambda, \mu)$ with $\mu > Y$, let $L' = (\lambda, Y)$ be the parallel slice with origin at $(0, Y)$. All critical values of $F^{1,2}$ are below L and L' by construction (recall that all critical points are assumed in the upper-right quadrant), hence we obtain the push by projecting vertically upwards. Looking at the second row in Table 4.1, we see that the weighted pushes are independent of μ, and therefore equal for L and L'. Hence, the weighted bottleneck distances along L and L' are equal: $W_\infty(F^1_L, F^2_L) = W_\infty(F^1_{L'}, F^2_{L'})$. See Figure 4.3 for an illustration. We conclude that for y-slices it suffices to consider $0 \leqslant \mu \leqslant Y$ in (4.1) without changing the matching distance. An analogous argument shows that for x-slices, it is only necessary to consider $0 \leqslant \mu \leqslant X$.

Summarizing the last two observations, we arrive at the following statement. There are sets $\mathcal{L}_1$ of flat x-slices, $\mathcal{L}_2$ of steep x-slices, $\mathcal{L}_3$ of flat y-slices, and $\mathcal{L}_4$ of steep y-slices (with each set containing some vertical/horizontal lines

as limit case) such that

$$\mathrm{d_M}(F^1, F^2) = \sup_{L \in \mathcal{L}_1 \cup \ldots \cup \mathcal{L}_4} W_\infty(F^1_L, F^2_L) \tag{4.2}$$

and such that $\mathcal{L}_1$ and $\mathcal{L}_2$ are parameterized by the rectangle $[0,1] \times [0,X]$ and $\mathcal{L}_3$ and $\mathcal{L}_4$ by $[0,1] \times [0,Y]$. Note that all these boxes are compact, which means that the supremum becomes a maximum.

Approximation. We present an approximation algorithm that, given two bi-filtrations F^1 and F^2 and some $\varepsilon > 0$ returns a number δ such that

$$\mathrm{d_M}(F^1, F^2) - \varepsilon \leqslant \delta \leqslant \mathrm{d_M}(F^1, F^2).$$

We assume that the two bi-filtrations are given as simplicial complexes, i.e., a list of simplices, where each simplex is annotated with two real values denoting the critical value of the simplex. In the description, we set $T := \{x\text{-flat}, x\text{-steep}, y\text{-flat}, y\text{-steep}\}$ for the type of a slice. The algorithm is based on the following two primitives:

`Eval`(F^1, F^2, L) Computes $W_\infty(F^1_L, F^2_L)$, where L is specified by the triple (λ, μ, t) where (λ, μ) are the parameterization of L and $t \in T$ denotes its type.

`Bound`(F^1, F^2, B, t) If B is an axis-parallel rectangle and $t \in T$, the pair (B, t) specifies a set of slices $\mathcal{L}_0$. The primitive computes a number $\mu \in \mathbb{R}$ such that

$$W_\infty(F^1_L, F^2_L) \leqslant \mu$$

for every $L \in \mathcal{L}_0$.

With these two primitives, we can state our approximation algorithm: from now on, we refer to axis-parallel rectangles as *boxes* for brevity. We start by computing maximal coordinates X and Y of critical values of F^1 and F^2 and enqueueing the four initial items $([0,1] \times [0,Y], y\text{-steep})$, $([0,1] \times [0,Y], y\text{-flat})$, $([0,1] \times [0,X], x\text{-steep})$, and $([0,1] \times [0,X], x\text{-flat})$ into a FIFO-queue. We also maintain a variable ρ storing the largest bottleneck distance encountered so far, initialized to 0.

Now, we pop items from the queue and repeat the following steps: for an item (B, t), let L denote the slice that corresponds to the center point of

B. We call $\texttt{Eval}(F^1, F^2, L)$ and update ρ if the computed value is bigger than the current maximum. Then, we compute $\mu \leftarrow \texttt{Bound}(F^1, F^2, B, t)$. If $\mu > \rho + \varepsilon$, we split B into 4 sub-boxes $B_1, \ldots, B_4$ of equal dimensions (using the center as splitpoint) and enqueue $(B_1, t), \ldots, (B_4, t)$. When the queue is empty, we return $\delta \leftarrow \rho$. This ends the description of the algorithm.

Assuming that the above algorithm terminates (which is unclear at this point because it depends on the implementation of the `Bound` primitive), we claim that the returned value is indeed an ε-approximation.

Proof. We need to show that the result of the approximation algorithm δ satisfies the following inequalities:

$$\mathrm{d_M}(F^1, F^2) - \varepsilon \leqslant \delta \leqslant \mathrm{d_M}(F^1, F^2).$$

Firstly, the value ρ is non-decreasing during the algorithm. Call a box *terminal* if it does not get subdivided during the algorithm (in other words, considering the set of boxes processed in the algorithm as a *quad-tree*, terminal boxes correspond to leaves in the quad-tree). By construction, we have that for each terminal box and each slice L specified by the box

$$W_\infty(F^1_L, F^2_L) \leqslant \mu \leqslant \rho + \varepsilon \leqslant \delta + \varepsilon$$

Moreover, the terminal boxes form a cover of the initial boxes; more precisely, the union of all terminal boxes with type y-steep is equal to $[0,1] \times [0, Y]$, and similarly for the other three types. Note that the initial boxes correspond to the sets $\mathcal{L}_1, \ldots, \mathcal{L}_4$ from (4.2). Hence, writing $\mathcal{T}$ for the set of all slices contained in a terminal box, we have that $\mathcal{T} = \mathcal{L}_1 \cup \ldots \cup \mathcal{L}_4$ and it follows that

$$\begin{aligned} \mathrm{d_M}(F^1, F^2) &= \sup_{L \in \mathcal{L}_1 \cup \ldots \cup \mathcal{L}_4} W_\infty(F^1_L, F^2_L) \\ &= \sup_{L \in \mathcal{T}} W_\infty(F^1_L, F^2_L) \leqslant \delta + \varepsilon \end{aligned}$$

proving the first inequality. On the other hand, since ρ is always a bottleneck distance for some slice, we have that $\rho \leqslant \mathrm{d_M}(F^1, F^2)$ throughout the algorithm, hence also $\delta \leqslant \mathrm{d_M}(F^1, F^2)$. □

A variant of the above algorithm computes a relative approximation of the matching distance, that is, a number δ such that

$$\mathrm{d_M}(F^1, F^2) \leqslant \delta \leqslant (1+\varepsilon)\,\mathrm{d_M}(F^1, F^2).$$

The algorithm is analogous to the above, with the difference that a box is subdivided if $\mu > (1+\varepsilon)\rho$, and at the end of the algorithm $(1+\varepsilon)\rho$ is returned as δ. The correctness of this variant follows similarly. However, as we will see at the end of Section 4.4, the algorithm terminates only if $\mathrm{d_M}(F^1, F^2) > 0$, and its complexity depends on the value of the matching distance.

What is needed to realize the `Eval` primitive? First, we have to compute the weighted pushes of each critical value of F^1 and of F^2, which we can do using Lemma 4.1 in time proportional to the number of critical values. Then, we have to compute the persistence diagrams of F^1 and of F^2, and compute their bottleneck distance. Both steps are well-studied standard tasks in persistent homology, and several practically efficient algorithms have been studied. We use Phat [12] for computing persistence diagrams and Hera [72] for the bottleneck computation.

4.4 The `Bound` primitive

Recall that the input of `Bound` is (F^1, F^2, B, t), where (B, t) specifies a collection of slices of type t. In what follows, we will identify points in B with the parameterized slice, writing $L \in B$ to denote that L is obtained from a pair of parameters $(\lambda, \mu) \in B$ with respect to type t (which we skip for notational convenience).

Let L_c be the slice corresponding to the center of B. Now define the *variation* of a point $p \in \mathbb{R}^2$ for B as

$$v(p, B) := \max_{L \in B} |\,\mathrm{wpush}_p(L) - \mathrm{wpush}_p(L_c)|,$$

where $\mathrm{wpush}_p(L) = \mathrm{wpush}(p, L)$. In words, the variation denotes how much the weighted push of the point p can change when the slice is changed

within the box B. For a bi-filtration F, we define

$$v(F, B) := \max_{p \text{ critical value of } F} v(p, B).$$

The next lemma shows that the variation yields an upper bound for the bottleneck distance within a box.

Lemma 4.2. *With the notation as before, we have that for two filtrations F^1, F^2 that*

$$\sup_{L \in B} W_\infty(F^1_L, F^2_L) \leqslant v(F^1, B) + W_\infty(F^1_{L_c}, F^2_{L_c}) + v(F^2, B)$$

Proof. By triangle inequality of the bottleneck distance,

$$W_\infty(F^1_L, F^2_L) \leqslant W_\infty(F^1_L, F^1_{L_c}) + W_\infty(F^1_{L_c}, F^2_{L_c}) + W_\infty(F^2_{L_c}, F^2_L).$$

Looking at the first term on the right, we have two filtrations of the same simplicial complex, and every critical values changes by at most $v(F^1, B)$ by definition of the variation. Hence, by stability of the bottleneck distance,

$$W_\infty(F^1_L, F^1_{L_c}) \leqslant v(F^1, B).$$

The same argument applies to the third term which proves the theorem. □

Note that the second term in the bound of Lemma 4.2 is the value at the center slice, which is already computed in the algorithm. It remains to compute the variation of a bi-filtration within B. This, in turn, we do by analyzing the variation of a point p within B. We show

Theorem 4.3. *For a box B, let $L_1, \ldots, L_4$ be the four slices on the corners of B. Then*

$$v(p, B) = \max_{i=1,\ldots,4} |\operatorname{wpush}_p(L_i) - \operatorname{wpush}_p(L_c)|$$

The theorem gives a direct algorithm to compute $v(p, B)$, just by computing the weighted pushes at the four corners (in constant time) and return the maximal difference to the weighted push in the center. Doing so for every critical point of a bi-filtration F yields $v(F, B)$, and with Lemma 4.2 an algorithm for the `Bound` primitive that runs in time proportional to the

number of critical points of F^1 and F^2. We refer to this bound as *local linear bound* (where the term "linear" refers to the computational complexity), or as *L-bound*.

To prove the theorem, it will be convenient to define

$$D(\lambda, \mu) := |\operatorname{wpush}_p(L_{\lambda,\mu}) - \operatorname{wpush}_p(L_c)|,$$

where $L_{\lambda,\mu}$ is the slice corresponding to (λ, μ). It follows immediately that

$$v(p, B) = \max_{(\lambda,\mu)\in B} D(\lambda, \mu).$$

and the theorem states that D is maximized in one of the corners of B. The next lemma reveals the structure of D along vertical and horizontal line segments within B.

Lemma 4.4. *Let $B = [a, b] \times [c, d]$. For any $\lambda_0 \in [a, b]$, the function*

$$D(\lambda_0, \cdot) : [c, d] \to \mathbb{R}$$

is maximized at c or at d. Likewise, for μ_0 fixed, the function

$$D(\cdot, \mu_0) : [a, b] \to \mathbb{R}$$

is maximized at a or at b.

Proof. First, we briefly sketch the argument for $D(\lambda_0, \cdot)$. For all four types of B, the function $D(\lambda_0, \mu)$ is either of the form $|s\mu + t|$ for constants $s, t \in \mathbb{R}$, a constant function, or a continuous combination of the two forms, depending on the location of p with respect to the slices (λ_0, μ) with $\mu \in [c, d]$. In either case, the function has no isolated local maxima, and therefore attains its maximum over $[a, b]$ on the boundary.

Throughout the proof, we write $w := \operatorname{wpush}_p(L_c)$, which is a constant independent of λ or μ. Let us consider the case of flat y-slices first. For $\lambda_0 \in [a, b]$ fixed, write L_{top} for the slice (λ_0, d) and L_{bottom} for the slice (λ_0, c). There are three possible locations of the point p: it can be above both L_{bottom} and L_{top}, below both of them, or above L_{bottom} and below L_{top}.

In the first case, D takes the form

$$D(\lambda_0, \mu) = |p_y - \mu - w|$$

on the whole interval $[c, d]$ (cf. Table 4.1). This is a "V-shaped" function which takes its maximum at a boundary point.

In the second case, D takes the form

$$D(\lambda_0, \mu) = |\lambda_0 p_x - w|$$

which is a constant function, clearly also being maximal at either boundary point.

In the third case, there is a unique point $\xi \in [c, d]$ such that p lies on the slice parameterized by (λ_0, ξ). Then, on the interval $[c, \xi]$, D is a V-shaped function as above, and on $[\xi, d]$, D is a constant function (and the two branches coincide at ξ, since D is continuous). It follows that also in this case, $D(\lambda_0, \cdot)$ is maximized at a boundary point.

The analysis of the function $D(\cdot, \mu_0)$ for $\mu_0 \in [c, d]$ is very similar. We write L_{top} for the slice (b, μ_0) and L_{bottom} for (a, μ_0). Note that the slices (λ, μ_0) for $\lambda \in [a, b]$ correspond to the slices obtained when rotating from L_{bottom} to L_{top} in counterclockwise direction with fixed origin $(0, \mu_0)$. There are same three possible cases as above for the location of p with respect to L_{top} and L_{bottom}.

If p is above both slices, D takes the form

$$D(\lambda, \mu_0) = |p_y - \mu_0 - w|$$

on $[a, b]$, which is a constant function. If p is below both slices, D takes the form

$$D(\lambda, \mu_0) = |\lambda p_x - w|$$

which is a V-shaped function. If p is in-between the slices, there is again a unique ξ such that p lies on the slice (ξ, μ_0), and the function splits into a V-shaped branch and a constant branch. This proves the statement for the case of flat y-slices.

The other three cases are analogous: In all of them, the functions $D(\lambda_0, \cdot)$ and $D(\cdot, \mu_0)$ are either V-shaped, constant, or a combination of both. □

Proof. (of Theorem 4.3) Let (λ, μ) be any point in B. By Lemma 4.4 we can move (λ, μ) vertically to either the lower or upper boundary without decreasing the D-value. Then, using Lemma 4.4, we can move the point horizontally to one the corners, again without decreasing the D-value. The statement follows. □

A coarser bound. We have derived a method to compute $v(p, B)$ exactly which takes linear time. Alternatively, we can derive an upper bound as follows:

Theorem 4.5. *Let B be a box $[\lambda_{\min}, \lambda_{\max}] \times [\mu_{\min}, \mu_{\max}]$ with center (λ_c, μ_c), width $\Delta\lambda = \lambda_{\max} - \lambda_{\min}$ and height $\Delta\mu = \mu_{\max} - \mu_{\min}$. Then, for any point $p \in [0, X] \times [0, Y]$, $v(p, B)$ is at most*

$$\begin{cases} \frac{1}{2}(\Delta\mu + X\Delta\lambda) & \text{for flat } y\text{-slices} \\ \frac{1}{2}(\lambda_c\Delta\mu + (Y - \mu_{\min})\Delta\lambda)\} & \text{for steep } y\text{-slices} \\ \frac{1}{2}(\lambda_c\Delta\mu + (X - \mu_{\min})\Delta\lambda) & \text{for flat } x\text{-slices} \\ \frac{1}{2}(\Delta\mu + Y\Delta\lambda) & \text{for steep } x\text{-slices}. \end{cases}$$

Importantly, the bound is independent of p, and hence also an upper bound for $v(F, B)$ that can be computed in constant time; we refer to it as *local constant bound* or *C-bound*.

Proof. The proof of Theorem 4.5 is based on deriving a bound of how much $\mathrm{wpush}_p(L)$ and $\mathrm{wpush}_p(L')$ can differ for two slices $L = (\lambda, \mu)$ and $L' = (\lambda', \mu')$ in dependence of $|\lambda - \lambda'|$ and $|\mu - \mu'|$. This bound, in turn, is derived separately for all four types of boxes and involves an inner case distinction depending on whether p lies above both slices, below both slices, or in-between. In either case, the claim of the statement follows from the bound by plugging in the center slice of a box for either L or L'.

Let us start with the case of flat y-slices. We first prove that for any two slices $L = (\lambda, \mu)$ and $L' = (\lambda', \mu')$ and any $p \in [0, X] \times [0, Y]$, it holds that

$$|\,\mathrm{wpush}_p(L) - \mathrm{wpush}_p(L')| \leqslant |\mu - \mu'| + X|\lambda - \lambda'|$$

We consider three cases: if p is above both L and L',

$$|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(L')| = |(p_y + \mu) - (p_y + \mu')| = |\mu - \mu'|,$$

and the bound clearly holds. If p is below both slices,

$$|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(L')| = |\lambda p_x - \lambda' p_x| \leqslant p_x|\lambda - \lambda'|,$$

and the bound holds, because X is the maximal possible value of p_x. Finally, if p is above L and below L' (or vice versa), then the line segment connecting (λ, μ) and (λ', μ') contains at least one point $(\tilde{\lambda}, \tilde{\mu})$ such that the slice $\tilde{L}$ defined by these parameter values contains the point p. Since p is on $\tilde{L}$, we can use either formula for the weighted push in the left column of Table 4.1 for $\tilde{L}$. Together with the triangle inequality, we obtain:

$$\begin{aligned}
&|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(L')| \\
=&|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(\tilde{L})| + |\operatorname{wpush}_p(\tilde{L}) - \operatorname{wpush}_p(L')| \\
=&|(p_y - \mu) - (p_y - \tilde{\mu})| + |\tilde{\lambda} p_x - \lambda p_x| \\
\leqslant&|\tilde{\mu} - \mu| + p_x|\tilde{\lambda} - \lambda'| \\
\leqslant&|\mu' - \mu| + X|\lambda' - \lambda|,
\end{aligned}$$

where the last inequality holds, because $\tilde{\mu}$ is between μ and μ', and $\tilde{\lambda}$ is between λ and λ'.

The first case of the statement follows at once by setting L' to be the center of the box B, and L to be any point in B since in this case, $|\lambda' - \lambda| \leqslant \Delta\lambda/2$ and $|\mu' - \mu| \leqslant \Delta\mu/2$.

Next we consider the case of steep y-slices. We claim that for any two slices $L = (\lambda, \mu)$ and $L' = (\lambda', \mu')$ and any $p \in [0, X] \times [0, Y]$, it holds that

$$\begin{aligned}
&|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(L')| \\
\leqslant&\lambda|\mu - \mu'| + (Y - \mu')|\lambda - \lambda'|
\end{aligned}$$

We consider the same three cases as above. If p is below both L and L', the weighted pushes are both equal to p_x, and the difference is 0. If p is above

both L and L', we calculate

$$\begin{aligned}
&|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(L')| \\
=&|\lambda(p_y - \mu) - \lambda'(p_y - \mu')| \\
=&|\lambda(p_y - \mu) - \lambda(p_y - \mu') + \lambda(p_y - \mu') - \lambda'(p_y - \mu')| \\
\leqslant&|\lambda(\mu' - \mu)| + |(\lambda - \lambda')(p_y - \mu')| \\
=&\lambda|\mu' - \mu| + (p_y - \mu')|\lambda - \lambda'|.
\end{aligned}$$

Note that in the last line, we use that $p_y - \mu'$ is positive, which follows from p being above L', and $p_x \geqslant 0$.

In the third case, p is above L and below L'. Instead of the line segment connecting them, we consider the path from (λ, μ) to (λ', μ') that first goes vertically to (λ, μ'), and then horizontally to (λ', μ'). Also on this path, there is a slice $\tilde{L}$ such that p is on $\tilde{L}$. Then, by triangle inequality, the difference of the weighted pushes for L and L' is at most

$$|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(\tilde{L})| + |\operatorname{wpush}_p(\tilde{L}) - \operatorname{wpush}_p(L')|$$

and the second term is equal to 0. Hence, using the previous calculation, the difference can be bounded by

$$\lambda|\tilde{\mu} - \mu| + (p_y - \tilde{\mu})|\lambda - \tilde{\lambda}|$$

Now, if $\tilde{L}$ is on the vertical branch of the path, $\lambda = \tilde{\lambda}$, and the second term vanishes. If $\tilde{L}$ is on the horizontal part, $\tilde{\mu} = \mu'$, and the bound above is equal to

$$\lambda|\mu' - \mu| + (p_y - \mu')|\lambda - \tilde{\lambda}|$$

and the bound follows because $|\lambda - \tilde{\lambda}| \leqslant |\lambda - \lambda'|$.

Using this estimate on $|\operatorname{wpush}_p(L) - \operatorname{wpush}_p(L')|$, the theorem statement for steep y-slices follows by choosing L as the center slice, and L' as any other slice in B. Note that using choosing L' as the center slice instead yields the (seemingly) different bound

$$\frac{1}{2}\left(\lambda_{\max}\Delta\mu + (Y - \mu_c)\Delta\lambda\right)\},$$

but it can be verified by a simple calculation that both bounds are equal. The bounds for x-slices are proved analogously. □

Termination and complexity. We show next that our absolute approximation algorithm terminates when realized with either the local linear bound or the local constant bound. In what follows, set $C := \max\{X, Y\}$. In the subdivision process, each box B considered is assigned a *level*, where the level of the four initial boxes is 0, and the four sub-boxes obtained from a level-k-box have level $k+1$. Since every box is subdivided by the center, we have immediately that for a level-k-box, $\Delta\lambda = 2^{-k}$, and $\Delta\mu \leqslant C2^{-k}$. Using these estimates in Theorem 4.5, we obtain

$$v(F^i, B) \leqslant \frac{1}{2}(C2^{-k} + C2^{-k}) = C2^{-k}. \tag{4.3}$$

for $i = 1, 2$ and every level-k-box B considered by the algorithm. Note that the local constant bound yields a bound on $v(F^i, B)$ that is not worse, and so does the local linear bound (which computes $v(F^i, B)$ exactly). Hence we have

Lemma 4.6. *Let B be a level-k-box considered in the algorithm. Then, μ, the result of the* `Bound` *primitive in the algorithm, satisfies*

$$\mu \leqslant W_\infty(F^1_{L_c}, F^2_{L_c}) + 2C2^{-k}$$

both for the local linear and local constant bound.

We can now easily derive a maximal level k such that no box on level k gets subdivided:

Lemma 4.7. *A level-k-box with*

$$k := \left\lceil \log \frac{2C}{\varepsilon} \right\rceil$$

(where the logarithm is with base 2) does not get subdivided by the algorithm.

Proof. We have to show that $\mu \leqslant \rho + \varepsilon$, where μ is the upper bound computed by the `Bound` primitive, and ρ is largest weighted bottleneck distance encountered at the moment when the algorithm decide whether to subdivide B. Note that in this moment, $\rho \geqslant W_\infty(F^1_{L_c}, F^2_{L_c})$ is ensured because

the latter value has been computed in the previous step. Hence, using the previous lemma,

$$\mu \leqslant W_\infty(F^1_{L_c}, F^2_{L_c}) + 2C2^{-k} \leqslant \rho + 2C2^{-k} \leqslant \rho + \varepsilon,$$

where the last step follows from the choice of k. □

Theorem 4.8. *Our algorithm to compute an absolute ε-approximation terminates in*

$$O(n^3 \left(\frac{C}{\varepsilon}\right)^2)$$

steps in the worst case (both for the linear and constant bound).

Proof. With k as above, the worst case is that the algorithm considers all boxes of level k. In that case, the total number of boxes considered is

$$4(4^0 + 4^1 + \ldots + 4^k = O(4^k)$$

Plugging in k yields that $4^k = (2^k)^2 = O(\left(\frac{C}{\varepsilon}\right)^2)$ as the number of considered boxes. On each box, the algorithm evaluates the weighted bottleneck distance at the center slice, which requires the computation of weighted pushes, of two persistence diagrams, and of their bottleneck distance. Complexity-wise, the dominating step is the persistence computation, which we do in $O(n^3)$ steps (this complexity can be reduced to $O(n^\omega)$, where ω is the matrix multiplication constant [86]). The complexity bound follows. □

A similar bound can be derived for the variant of computing a relative approximation.

Theorem 4.9. *If $\mathrm{d_M}(F^1, F^2) > 0$, our algorithm to compute a relative $(1+\varepsilon)$-approximation terminates in*

$$O(n^3 \left(\frac{C(1+\varepsilon)}{\varepsilon\, \mathrm{d_M}(F^1, F^2)}\right)^2)$$

steps in the worst case.

For the proof, we need the following lemma stating that ρ will eventually be a close approximation of the matching distance.

Lemma 4.10. *Write* $\mathrm{d_M} := \mathrm{d_M}(F^1, F^2)$ *and assume* $\mathrm{d_M} > 0$. *For*

$$k \geqslant 1 + \left\lceil \log \frac{(1+\varepsilon)2C}{\varepsilon \, \mathrm{d_M}} \right\rceil,$$

it holds that when the algorithm considers a level-k-box, we have that $\rho \geqslant \mathrm{d_M} / (1+\varepsilon)$.

Proof. Let L^* be a slice such that the matching distance is realized as $W_\infty(F^1_{L^*}, F^2_{L^*})$. Note that the algorithm handles all boxes of level $< k$ before handling any box of level k. Now, we distinguish two cases:

The first case is that L^* lies in some box of level $< k$ for which the algorithm did not subdivide further. That means that $\mu \leqslant (1+\varepsilon)\rho'$, where μ is an upper bound for the box and ρ' is the value of ρ at this moment of the algorithm. Note that since L^* lies in B, $\mu \geqslant \mathrm{d_M}$ must hold. Also, $\rho' \leqslant \rho$ because ρ only increases. It follows that

$$\mathrm{d_M} \leqslant \mu \leqslant (1+\varepsilon)\rho' \leqslant (1+\varepsilon)\rho$$

proving the statement for the first case.

The second case is that L^* lies in some level-$(k-1)$-box B which has been subdivided. By the way how μ is computed, we have that

$$\mu \leqslant W_\infty(F^1_{L_c}, F^2_{L_c}) + 2C2^{-(k-1)},$$

where L_c is the center slice of the box. Moreover, as before, $\mathrm{d_M} \leqslant \mu$ holds, and $W_\infty(F^1_{L_c}, F^2_{L_c}) \leqslant \rho$ because ρ is updated using L_c. In summary, we obtain

$$\mathrm{d_M} \leqslant \rho + 2C2^{-(k-1)}$$

The bound on k ensures that

$$2C2^{-(k-1)} \leqslant 2C2^{\log \frac{\varepsilon \, \mathrm{d_M}}{2C(1+\varepsilon)}} = \frac{\varepsilon \, \mathrm{d_M}}{1+\varepsilon}$$

so we obtain

$$\rho \geqslant \mathrm{d_M} - 2C2^{-(k-1)} \geqslant \mathrm{d_M} - \frac{\varepsilon\,\mathrm{d_M}}{1+\varepsilon} = \frac{\mathrm{d_M}}{1+\varepsilon}.$$

□

Lemma 4.11. *If* $\mathrm{d_M} > 0$*, a level-k-box with*

$$k := 1 + \left\lceil \log \frac{(1+\varepsilon)2C}{\varepsilon\,\mathrm{d_M}} \right\rceil$$

does not get subdivided by the relative approximation algorithm.

Proof. Let B be some level-k-box. We have to show that $\mu \leqslant (1+\varepsilon)\rho$ for B. Note that

$$\mu \leqslant W_\infty(F^1_{L_c}, F^2_{L_c}) + 2C2^{-k} \leqslant \rho + \frac{\varepsilon\,\mathrm{d_M}}{(1+\varepsilon)}$$

Moreover, the k in question satisfies the assumptions of Lemma 4.10, so we have that $\mathrm{d_M} \leqslant (1+\varepsilon)\rho$, and we obtain

$$\mu \leqslant \rho + \frac{\varepsilon\,\mathrm{d_M}}{(1+\varepsilon)} \leqslant (1+\varepsilon)\rho$$

□

Proof. (of Theorem 4.9) The proof is analogous to the proof of Theorem 4.8, noting that the algorithm has to consider

$$O(4^k) = O(\left(\frac{(1+\varepsilon)2C}{\varepsilon\,\mathrm{d_M}}\right)^2)$$

boxes, and the cost for each box is $O(n^3)$, as follows from the previous lemmas. □

4.5 Experiments

Experimental setup. Our experiments were performed on a workstation with an Intel(R) Xeon(R) CPU E5-1650 v3 CPU (6 cores, 3.5GHz) and 64 GB RAM, running GNU/Linux (Ubuntu 16.04.5). The code was written in C++ and compiled with gcc-8.1.0.

We generated two datasets, which we call **GH** and **ED**, following [20]. Unfortunately, we were unable to get either the code or the data used by the authors. Each of the 70 files in the datasets is a lower-star bi-filtration of a triangular mesh (2-dimensional complex), representing a 3D shape. We also generated dataset **RND** of larger random bi-filtrations with up to 2,000 vertices. We give a more detailed description of the datasets in the next paragraph. In all our experiments we used persistence diagrams in dimension 0. In the experiments with the datasets **GH** and **ED**, we computed all pairwise distances; in the experiments with **RND** we computed distances only between bi-filtrations with the same number of vertices. We used the relative error threshold, which we call ε in this section throughout (i.e., we always compute $1+\varepsilon$-approximation).

Datasets.

Lower-star bi-filtrations. Both datasets **GH** and **ED** are based on a Non-Rigid World Benchmark [27], a collection of 3D-shapes represented as triangular meshes. A triangular mesh is a geometric realization of a 2-dimensional simplicial complex, and, if we fix a function $\varphi\colon V \to \mathbb{R}^2$ on the vertices of a mesh, this gives rise to the lower-star bi-filtration (defined at the end of Section 2.5).

For **GH**, we use the function $\varphi^{GH} = (\varphi_1^{GH}/K_1, \varphi_2^{GH}/K_2)$ defined as fol-

lows:

$$\varphi_1^{GH}(v) = \text{integral geodesic distance of } v; \tag{4.4}$$
$$\varphi_2^{GH}(v) = \text{HKS at } t = 1000; \tag{4.5}$$
$$K_1 = \max_v \varphi_1^{GH}(v); \tag{4.6}$$
$$K_2 = \max_v \varphi_2^{GH}(v). \tag{4.7}$$

HKS in the formula for φ_2^{GH} stands for Heat Kernel Signature. It is computed by solving the discrete analogue of the heat equation on smooth manifolds (where the discrete Laplacian replaces the Laplace-Beltrami operator). The heat kernel was introduced in [99], and became a very popular tool in shape analysis; we used the publicly available code from [101] to compute it.

Let $s(v, w)$ be the length of the shortest path that connects v and w on the mesh and does not go through the interior of any of the mesh triangles. The integral geodesic distance $\varphi_1^{GH}(v)$ is a weighted sum over vertices $w \neq v$ of $s(v, w)$; it was introduced in [67], and we refer the reader to this paper for further details and motivation. We compute $s(v, w)$ using Dijkstra's algorithm on the graph that consists of the mesh vertices and edges with weight of an edge being Euclidean distance between its endpoints.

Normalization constants K_1, K_2 ensure that the maximal coordinates X and Y are 1.

Let us now describe the dataset **ED**. We fix a mesh with vertices $V = \{v_1, \ldots, v_n\}$, and let b be its center of mass, $b := \frac{1}{n}\sum_{i=1}^n v_i$. Vector $\vec{w}$ is defined as

$$\vec{w} := \frac{\sum_{i=1}^n (v_i - b)\|v_i - b\|}{\sum_{i=1}^n \|v_i - b\|^2}.$$

Let $\pi_{\vec{w}}$ be the plane passing through w and orthogonal to $\vec{w}$, let $\ell_{\vec{w}}$ be the line passing through w and parallel to $\vec{w}$; let $d(v, \ell)$ and $d(v, \pi)$ denote Euclidean distances from the point v to the line ℓ and plane π. The critical values of

ED bi-filtrations are given by the function $\varphi^{ED} = (\varphi_1^{ED}/K_1, \varphi_2^{ED}/K_2)$:

$$\varphi_1^{ED}(v_i) := 1 - \frac{d(v_i, \ell_{\vec{w}})}{\max_{k=1,\dots,n} d(v_k, \pi_{\vec{w}})}; \tag{4.8}$$

$$\varphi_2^{ED}(v_i) := 1 - \frac{d(v_i, \pi_{\vec{w}})}{\max_{k=1,\dots,n} d(v_k, \pi_{\vec{w}})}; \tag{4.9}$$

$$K_1 := \max_v \varphi_1^{ED}(v); \tag{4.10}$$

$$K_2 := \max_v \varphi_2^{ED}(v). \tag{4.11}$$

These formulas can be found on page 1743, Section 4.3 of [20]. The **ED** dataset can be easily computed, since the formulas involve only elementary geometric calculations.

We generated 70 bi-filtrations that cover different classes of the benchmark (male and female figures in different poses, seahorses, cats, etc), hence we have $2,415$ pairs to test.

Note that **GH** and **ED** are different from [20], because the authors also applied additional transformations to the meshes before computing the bi-filtrations. We skipped this step, because it is not clearly described and hard to reproduce.

Random bi-filtrations. In order to see the scaling on larger inputs, we generated random bi-filtrations as follows. The input parameters are the number of maximal simplices M, total number of vertices N, and the dimension of maximal simplices D. We randomly generated M distinct subsets of cardinality $D+1$ of the set $\{1,\dots,N\}$, and for each of these maximal simplices σ we chose the x and y coordinates of $\varphi(\sigma)$ uniformly at random from $[0,1000]$. Assume now that we defined φ on all simplices in some dimension $d+1 \leqslant D$, and we want to define $\varphi(\tau)$ on a simplex τ of dimension d. We know that $\varphi(\tau)$ must appear in the bi-filtration before any of its co-faces. To ensure that, we intersect all rectangles with the bottom left corner at $(0,0)$ and the upper right corner at $\varphi(\sigma)$, where σ ranges over all simplices such that τ is a face of σ and $\varphi(\sigma)$ is defined. This intersection is itself a rectangle; we pick an integral point in this rectangle uniformly at random as $\varphi(\sigma)$. We generated 6 random bi-filtrations of dimension 1

for each $N \in \{500, 1000, 2000\}$, with $M = 4N$. These bi-filtrations form the dataset **RND**.

Comparison of different bounds. First, we experimentally compare the performance of our algorithm with the L-bound from Theorem 4.3 and with the C-bound from Theorem 4.5. Obviously, the L-bound is sharper, so it allows us to subdivide fewer boxes and in this sense is more efficient. However, it is not a priori clear that the L-bound is preferable, since its own computation takes $O(n)$ time per box, in contrast to the constant bound.

There are two natural optimizations of the algorithm with the L-bound. Note that the value returned by `Bound` is used only to decide whether we have to subdivide the current box: we subdivide, if the bound is less than $(1+\varepsilon)\rho$, where ρ is the lower bound, the maximum of all $W_\infty(F_L^1, F_L^2)$ we have computed so far. The first optimization is to compute the C-bound, and, if it is less than $(1+\varepsilon)\rho$, stop subdividing (the L-bound cannot be greater than the C-bound). Otherwise we range over all critical values p, computing their variation. As second optimization, we stop once we encounter a point p whose variation is large enough to ensure that the L-bound will be greater than $(1+\varepsilon)\rho$.

Secondly, we compare the local bounds L and C with a global bound, that is, a bound that depends only on the size of the box (a bound of that sort is used in [20]). This bound is provided by Equation (4.3), we call it the G-bound. [1]

Recall that the dominating step in the complexity analysis (and in practice) is the computation of persistence, and we perform two such computations when we call the `Eval` primitive. Therefore we are interested in the number of calls of `Eval`; for brevity, we refer to this number simply as the number of *calls*.

In Table 4.3, we give the average number of calls and timings for different datasets and values of ε. Actually, the variance behind the average in these tables is large, so we additionally provide Table 4.4 and Table 4.5, where we

[1] The constant factors in the G-bound are smaller than in the global bound derived in [20].

	#Calls			Time (min)		
	L	C	G	L	C	G
GH, $\varepsilon = 0.5$	938	2502	11082	2.08	3.67	18.78
ED, $\varepsilon = 0.1$	1455	3920	27529	2.58	3.26	25.96
ED, $\varepsilon = 0.5$	169	531	2112	0.28	0.42	1.67

Table 4.3: Average number of calls and average running time with the L-, C- and G-bounds for different datasets and relative error ε.

report the average, maximal, and minimal ratios of the number of calls and time that the algorithm needs with different bounds. For instance, the third line of Table 4.4 shows that for all pairs from **ED** that we tested with relative error $\varepsilon = 0.5$, switching from the local constant bound to the local linear bound reduces the number of calls by a factor between 1.78 and 4.92.

Table 4.5 shows that, as expected, the C-bound always performs better than the G-bound, with the average speedup around 2. The L-bound brings an additional speedup by a factor of 1.5-2 in terms of the running time; the number of calls is reduced more significantly, by a factor of 3. However, the second from the right column of Table 4.5 shows that the running time can sometimes moderately increase, if we switch to the L-bound from the C-bound. If we compare the G-bound with the L-bound directly (these numbers are not present in Table 4.5), the best speedup factor is 15.6, the worst one is 1.14, and the average is between 3 and 5, depending on the dataset.

Breadth-first search, depth-first search, and error decay. In the formulation of our algorithm we used a queue. This means that we traverse the quad-tree in breadth-first order: a box from level $k+1$ added to the queue will be processed only after all boxes from level k. It is possible to use a priority queue instead of a queue, and use level as priority to do a depth-first traversal. It is also possible to make a greedy algorithm, where the box with the highest value of the weighted bottleneck distance at its center is always picked first. We experimented with these variants and found no significant

	#Calls: G / C			#Calls: C / L		
Dataset, ε	Avg	Min	Max	Avg	Min	Max
GH, $\varepsilon = 0.5$	1.80	1.21	3.28	3.10	1.51	7.02
ED, $\varepsilon = 0.1$	2.93	1.43	5.07	3.00	1.88	6.82
ED, $\varepsilon = 0.5$	1.94	1.17	2.81	3.29	1.78	4.92
RND, $\varepsilon = 0.1$	6.06	2.76	10.58	2.08	1.91	2.47

Table 4.4: Comparison of number of calls between the global, local constant, and local linear bounds. G / C denotes the ratio of the G-bound compared with the C-bound; C / L denotes the ratio of the C-bound compared with the L-bound.

	Time: G / C			Time: C / L		
Dataset, ε	Avg	Min	Max	Avg	Min	Max
GH, $\varepsilon = 0.5$	1.66	1.00	3.18	2.03	0.75	6.32
ED, $\varepsilon = 0.1$	3.12	1.44	5.21	1.64	0.81	3.40
ED, $\varepsilon = 0.5$	2.08	1.07	3.38	1.59	0.92	3.89
RND, $\varepsilon = 0.1$	5.73	2.83	10.67	1.93	1.66	2.20

Table 4.5: Comparison of the running time between the global, local constant, and local linear bounds.

difference. The explanation is that a good lower bound is achieved after a small number of iterations in every variant, and the remaining part of the computations is mostly to certify the answer.

Nevertheless, we see a scenario where the priority queue formulation is reasonable. If we want to calculate the best possible bounds for the matching distance within a given time limit, it is easy to modify our algorithm for this case. We use the upper bound (the output of the `Bound` primitive) as priority. When the time is over, it suffices to peek at the top of the priority queue to get the current upper bound, and we can output the lower bound ρ and the relative error that we can guarantee at this point. It is instructive to plot how the relative error decreases as the algorithm runs. We provide one such plot in Figure 4.4. For instance, we can see that it takes approximately 3.5 times longer to bring the relative error below 0.1 than below 0.2, if we use the constant bound. This agrees with the complexity estimate in Theorem 4.9.

One detail in this plot is relevant for the experiments of the previous paragraph. If we choose a relative error ε_0 and draw a horizontal line $\varepsilon = \varepsilon_0$ in Figure 4.4 until it intersects the plotted curves, then the x-coordinate of the intersection is the time that our algorithm needs to guarantee a $1 + \varepsilon_0$ approximation with the corresponding bound. We can see that the difference between the time needed with the global bound and the time needed with the constant bound is not large for some values of ε_0, but a small change of ε_0 can rapidly increase it. Clearly, this is highly input-specific, and this partially explains the large variation in the improvement ratios that we observed above, when we ran experiments with fixed ε.

Reduction rate. The only measure reported in [20] is what the authors called the *reduction rate*. It is defined as $1 - c/4^k$, where c is the number of calls, k is the level of the deepest box in the quad-tree, on whose center the algorithm actually performs a call (i.e., computes the weighted bottleneck distance), and 4^k is the total number of boxes on level k. What does the reduction rate measure? Suppose that for some reason we decided to look at the k-th level of the quad-tree only. We can simply compute the weighted bottleneck distance at the center of each of the 4^k boxes and output the largest result; this would be a brute force approach. If we have a bound and guess the level k correctly, then we can guarantee the desired approximation

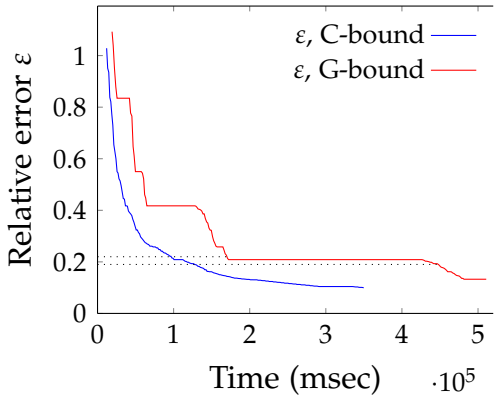


Figure 4.4: Error decay with time for the C- and G-bounds.

quality. The reduction rate shows which fraction of the 4^k calls we avoid by switching to the quad-tree algorithm; if it is 0.99, this means that we avoided 99% of calls.

We give our reduction rates (average, minimal, and maximal) in Table 4.6. The best reported reduction rate in [20] for $\varepsilon = 0.1$ is 94.7% (**GH**) and 93.8% (**ED**), with average values of 60.6% and 57%; minimal values are not provided. Our maximal values for the global bound approximately agree with theirs, and our average values are by roughly 10% higher.

This means that with the G-bound we can on average avoid not 57% of calls, as in [20], but 71%. This should be expected, because the G-bound has a smaller constant factor, see the end of Section 4.6. The advantage of the local and constant bounds becomes evident in the worst case; we never have reduction by less than 86% with the local bound, while the global bound can go as low as 30%.

Scaling on random bi-filtrations. In datasets **GH** and **ED**, the number of vertices is around 3,000. However, since these datasets are lower-star bi-filtrations, the cardinality of the persistence diagrams of the restrictions is smaller, typically between 20 and 60 points. In dataset **RND**, the cardinality of the diagrams in dimension 0 is almost equal to the number of vertices,

	Global			Local Constant		
	Max	Avg	Min	Max	Avg	Min
GH	92.95	69.47	56.43	96.95	86.97	75.77
RND	64.44	56.28	32.88	96.64	91.23	81.86
ED	97.45	71.00	30.78	98.01	89.07	69.35
	Local Linear					
	Max	Avg	Min			
GH	98.52	93.53	86.47			
RND	98.33	95.79	91.25			
ED	99.30	96.49	86.39			

Table 4.6: Reduction rate for $\varepsilon = 0.1$, in percents.

so it is larger by an order of magnitude even for 500 vertices. We use this dataset to show how our algorithm scales on larger inputs.

In Table 4.7, we see how the running time and the number of calls grows as we increase the size of the input. As in Table 4.3, the variance hidden behind the averaged numbers is large, but the bounds compare in the expected way: the linear bound outperforms the constant bound, and the constant bound outperforms the global bound.

	#Calls			Time (sec)		
#vert.	L	C	G	L	C	G
500	164	435	655	6.5	15.1	23.3
1000	893	2354	4222	197.2	505.7	983.5
2000	1233	3256	5995	911.8	2277.8	4254.3

Table 4.7: Average number of calls and average running time with bounds L, C, and G for random bifiltrations with different number of vertices. Relative error $\varepsilon = 0.5$.

	Time: G / C			Time: C / L		
#vert.	Avg	Min	Max	Avg	Min	Max
500	1.66	1.42	2.37	2.38	1.92	2.81
1000	1.76	1.37	2.22	2.53	2.34	2.77
2000	1.76	1.38	1.96	2.63	2.33	2.83

Table 4.8: Scaling on dataset **RND**, running time ratios for bounds G, C and L. Relative error $\varepsilon = 0.5$

In Table 4.8 and Table 4.9, we provide the ratios of the running time and the number of calls. These tables are similar to Table 4.5 and Table 4.4. For example, the first 3 columns of Table 4.8 were obtained as follows. For each pair of random bi-filtrations with the same number of vertices, we measure the time with the global bound and with the constant bound, and take their ratio. The average of these ratios is in the first column, the minimum is in the second column, and the maximum is in the third one.

Note that the ratios in these tables do not change much as we go from 500 vertices to 2,000. If we compare the constant bound and the linear bound, we notice that the ratios in Table 4.9 are very stable. The linear bound always reduces the number of calls by a factor of approximately 2.5. The running time ratios in Table 4.8 increase for larger inputs, getting closer to the ratios of the number of calls. This is actually expected, since the complexity of the `Eval` primitive is super-linear, so the time spent in `Eval` starts to subsume the time spent on computing the L bound. One should not expect that the ratio will grow with n: a better bound only reduces the number of calls of `Eval`, but cannot accelerate the computation of `Eval` itself.

Heatmaps. Recall that we have 4 rectangles $\mathcal{L}_1, \ldots, \mathcal{L}_4$ that parameterize 4 different types of slices. Each slice passing through the origin is at the same time an x-slice and a y-slice, and each slice with slope 1 is both flat and steep. Therefore we can glue the rectangles $\mathcal{L}_i$ together by identifying the points that represent the same slice, and we get a single domain $\mathcal{L}$ with the slice $y = x$ in the center, as in Figure 4.5 (the coordinates in this figure do

	#Calls: G / C			#Calls: C / L		
#vert.	Avg	Min	Max	Avg	Min	Max
500	1.60	1.28	2.22	2.67	2.52	2.78
1000	1.68	1.25	2.03	2.65	2.57	2.80
2000	1.76	1.34	2.02	2.61	2.48	2.74

Table 4.9: Scaling on dataset **RND**, #calls ratios for bounds G, C and L. Relative error $\varepsilon = 0.5$

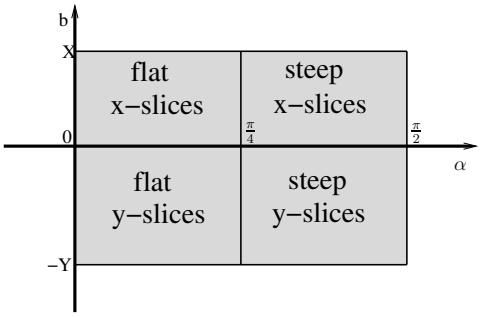


Figure 4.5: Decomposition of the domain $\mathcal{L}$ into 4 types.

not agree with the coordinates we use inside each $\mathcal{L}_i$ for parameterization in Section 4.3). We can visualize values of the weighted bottleneck distance by computing it at the center of each box of the quad-tree (of $\mathcal{L}$) on a fixed level, and using these values for the heatmap. The brighter a pixel of the heatmap, the larger the value of $W_\infty(F_L^1, F_L^2)$ for the corresponding slice L.

In Figure 4.6 and Figure 4.7, we show two examples (they were computed on the **ED** dataset) of heatmaps. For pair A, in Figure 4.6, high values of the weighted bottleneck distance are concentrated in a small bright spot around the center. We can expect that, whichever bound we use, the algorithm will need only a few subdivisions in the darker area to ensure that these boxes cannot improve the lower bound. The opposite is true for the heatmap of pair B, Figure 4.7, where a large part of all 4 quadrants has almost equal high values. The algorithm will need to subdivide the boxes covering this part until they become so small that the upper bound on each of them

will be within the error threshold. These expectations are confirmed by the experimental results. It takes 325 calls to approximate the matching distance with $\varepsilon = 0.1$ for pair A, and 838 calls for pair B, if we use the linear bound in both cases.

Pair B is also an example of a case when the local constant bound does not improve the performance in comparison with the global bound. Indeed, since for pair B we have $X = Y = 1$, the global bound and the local constant bound agree on two of the four quadrants (flat y-slices and steep x-slices, see Equation (4.3)), hence there will be no difference in the algorithm's behavior there, and in this example all four quadrants require many subdivisions.

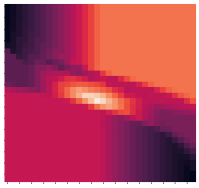


Figure 4.6: Heatmap A.

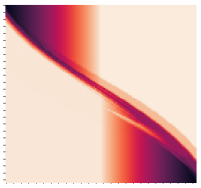


Figure 4.7: Heatmap B.

Additional plots. In Section 4.5, we plot the dependence of relative error on the number of calls for the local constant and local linear bounds. The plot is for the same input as Figure 4.4. Note that the plots are closer to each other than the plots for the global and the local constant bound in Figure 4.4, and, if we chose to use time as the x-axis, the difference would become even smaller.

In Section 4.5, we plot the evolution of the lower bound ρ and the upper bound μ (for the same input). Note that the lower bound stabilizes very early.

4.6 Conclusion

Comparison with [20]

Many ideas used in this chapter appear in [20] in some form. For instance, the idea of restricting the parameter space to a bounded region (see the discussion after our Lemma 4.1) corresponds to Lemma 3.1 in [20]. However, instead of completely disregarding the region outside of the bounded region, they introduce two points in this domain, which seems unnecessary and complicates the algorithmic description. As another example, our upper bound in theorem 4.5 corresponds to their bound from Lemma 3.3 (with worse constants, which we discuss below). The proof of their lemma contains a case distinction, where they consider, expressed in our notation, the case of two flat slices, two steep slices, and the mixed case of a flat and a steep slice. By our choice of splitting the parameter space in 4 parts, we can ensure that all slices within a box are of the same type, which makes the (tedious) analysis of mixed cases unnecessary.

Moreover, we improve on [20] in several algorithmic ways: foremost, our local bounds provide better estimates of the variation and lead to fewer subdivisions. Moreover, when we subdivide boxes, we keep the aspect ratio of the box the same in the next iteration. This allows us to cover the initial box with 4^k boxes of level k. The approach in [20] uses squares instead and hence requires $C4^k$ boxes on level k. Since their algorithm has to subdivide to a level of $O(C/\varepsilon)$ in the worst case (same as ours), the complexity becomes $O(n^3 \frac{C^3}{\varepsilon^2})$, which is a factor C worse. Finally, the approaches differ in the choice of the parameterization. Their approach, restated in geometric terms, represents a slice L by two parameters (λ, β), where λ is the sine of the angle of the L with the x-axis, and the origin is chosen as the point $(\beta, -\beta)$. While this approach has the pleasant effect of avoiding a case distinction between x-slices and y-slices, it has two downsides: first of all, the bounding rectangle in parameter space contains more slices than in our version, and the fact that the origin is further away from the critical points (which all lie in the upper-right quadrant) leads to a worsening of the bounds in Theorem 4.5. This partially explains the discrepancy of their upper bound

of $(16C + 2)\delta$ (Theorem 3.4 in [20]) and the bound of $4C\delta$ that could be achieved with our methods.

Generalizations.

We have restricted to the case of bi-filtrations in this work. Generalizations in several directions are possible. First of all, instead of bi-filtrations, our algorithm works the same when the input is a pair of presentations of persistence modules [79, 73, 23]. Since a minimal presentation of a bi-filtration can be of much smaller size than the bi-filtration itself, and its computation is feasible [79, 52], switching to a minimal presentation will most likely increase the performance further. We plan to investigate this in future work. Moreover, the case of k-critical bi-filtrations can be handled with our methodology, just by defining the push of a simplex as the minimal push over all its critical values. A careful inspection of our argument shows that this adaption does not lead to any complication. Finally, an extension of our approach to 3 and more parameters should be possible in principle, but we point out that the space of affine lines through $\mathbb{R}^d$ is $2(d-1)$ dimensional. Hence, already the next case of tri-filtrations requires a subdivision in $\mathbb{R}^4$, and it is questionable whether reasonably-sized instances could be handled by an extended algorithm.

As the experiments show, in some cases the cost of computing the L-bound makes the variant with the C-bound faster. However, parallelization of the L-bound is trivial, and we believe that on larger instances it will make the L-bound the best choice.

Finally, since the matching distance can be computed exactly in polynomial time [73], the question is whether there is a practical algorithm for this exact computation. Our current implementation can serve as base-line for a comparison between exact and approximate versions of matching distance computations.

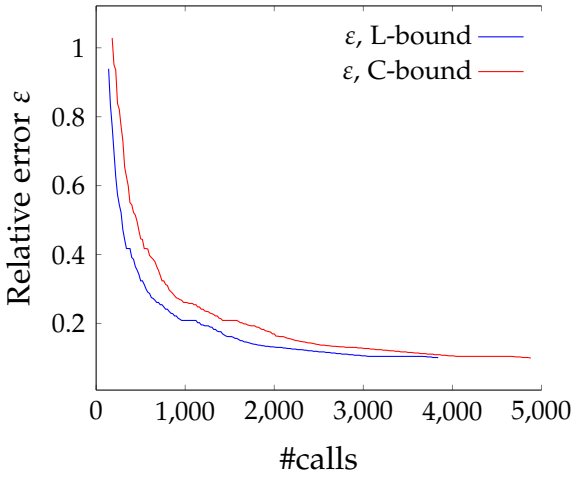


Figure 4.8: Error decay for C- and L-bounds.

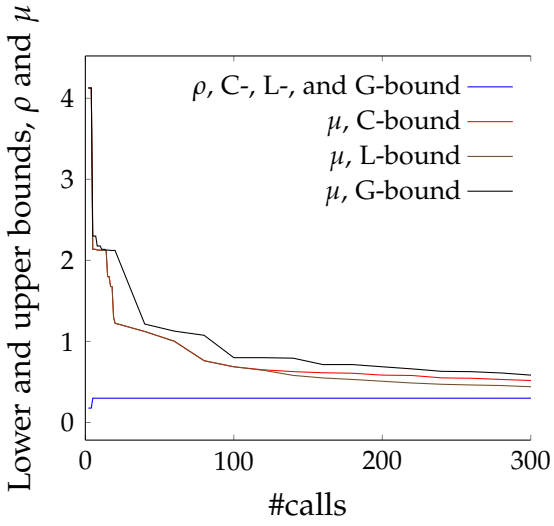


Figure 4.9: Lower and upper bounds (for C- and L-bounds).

5 Metric Spaces with Expensive Distances

This chapter is based on preprint [75].

5.1 Introduction

In the previous chapters we worked with concrete distances (bottleneck, Wasserstein, matching distance). While the implementations of our algorithms collected in Hera are used in practice ([91], [13]), the complexity of the algorithms still makes them prohibitive for sizable inputs. On the other hand, there exist special methods for computing the 0-th persistence, which are capable of processing fairly large filtrations. As for matching distance, if one wants a high-quality approximation, (with relative error less than 1 %), than the run time is measured in hours even for bi-filtrations of moderate size.

On the other hand, in applications it is typical to have not just a pair of topological descriptors to compare, but rather a sample of them. The two frequently appearing problems can be formalized as follows. Let $P := \{p_1, \dots, p_n\}$ be a set of n points in a metric space $(\mathcal{M}, \delta)$.

Approximate Nearest Neighbor Given $\varepsilon > 0$ and a point $q \in \mathcal{M}$, find $p_i \in P$ such that, for all $j = 1, \dots, n$,

$$\delta(q, p_i) \leqslant (1 + \varepsilon)\delta(q, p_j)$$

Spanner Given $\varepsilon > 0$, compute a weighted graph G with vertices in P such that for any $u, v \in P$, the shortest path distance between u and v is at most $(1 + \varepsilon)\delta(u, v)$.

While both these problems are extensively studied, the common assumption is that the cost C_δ of computing a distance in the metric space is a constant, and the performance of algorithms is expressed in terms of the number of points n. This assumption is motivated by the most natural setting, when $\mathcal{M} = \mathbb{R}^d$. In situations with expensive distance computations, it makes sense to study a different cost model, where only the number of distance computations is taken into account. For instance, that means that quadratic time operations in terms of n are not counted towards the time complexity, as long as these operations do not query any distance in $\mathcal{M}$. We also ignore the space complexity in our model.

We will restrict to the case of *doubling spaces*, that is, the doubling dimension of $\mathcal{M}$ is bounded by a constant. In that situation, standard constructions from computational geometry provide partial answers: Using net-trees [93], we can construct a ε-well-separated pair decomposition (WSPD) [32] using $O(n \log n)$ distance queries; a WSPD in turn yields a $(1+\varepsilon)$-spanner immediately. Net-trees can also be used to compute approximate nearest neighbors performing $O(\log n)$ distance computations per query point. Krauthgamer and Lee [76] investigated the *black box model*, and proved that ANN search for $\varepsilon < 2/5$ can be done efficiently (i.e., in polylogarithmic time, with polynomial preprocessing and space) if and only if the dimension is $O(\log \log n)$; their bounds count the number of distance computations. However, for our relaxed cost model, we pose the question whether simpler constructions achieve comparable, or even fewer distance computations.

We also propose a slight variant of our model: we assume that we also have access to an (efficient) 2-approximation algorithm for the distance queries. Queries to this approximation algorithm are not counted in the model, hence we can assume that for each pair of points (u, v), we know a number $A_{u,v}$ with $\delta(u,v) \leqslant A_{u,v} \leqslant 2\delta(u,v)$. This induces an approximate ordering of all distances in the metric space, and it is plausible to assume that such an ordering will simplify algorithmic tasks on metric spaces, at least in practice.

Contributions. We propose simple algorithms for spanner construction and approximate nearest neighbor search and evaluate them theoretically and experimentally in the defined cost model.

Our algorithms are based on the following simple idea: since distance computations are expensive and should be avoided, we try to obtain maximal information out of the distances computed so far. This information consists of lower and upper bounds for unknown distances, obtained from known distances by triangle inequality (see Figure 5.1). We remark that updating these bounds involves $\Omega(n^2)$ arithmetic operations whenever a new distance has been computed, turning the method useless in the standard computational model.

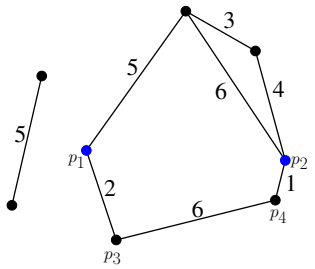


Figure 5.1: The computed distances are shown as edges in a graph. Note that the exact distance of p_1 and p_2 is unknown. The shortest path from p_1 to p_2 has length 9, which clearly constitutes an upper bound on the distance by triangle inequality. However, we can also infer that $\delta(p_1, p_2) \geqslant 3$: otherwise, the path from p_3 to p_4 via p_1 and p_2 would be shorter than the edge (p_3, p_4), again contradicting triangle inequality.

We propose several heuristics of how to explore the metric space to obtain accurate lower and upper bounds with a small number of distance computation. Once the ratio of upper and lower bound is at most $(1 + \varepsilon)$ for each point pair, the set of all computed distances forms the spanner. The experimentally most successful exploration strategy that we found is to repeatedly query the distance of a pair with the worst ratio of upper and lower bound. We call the obtained spanner the *blind greedy spanner*, as opposed to the well-known *greedy spanner* that precomputes all pairwise distances and only maintains upper bounds [3]. We demonstrate experimentally that on a collection of persistence diagrams of moderate size, computing a blind greedy spanner is faster than (approximately) computing all $\binom{n}{2}$ distances. Remarkably, we were not able to improve the quality when knowing initial 2-approximations of all point pairs. We also compare with a spanner construction based on WSPD. Our simple algorithms gives much smaller

spanners on the tested examples. Nevertheless, we leave the question open whether our construction yields a spanner of asymptotically linear size.

For approximate nearest neighbor, we devise a simple randomized incremental algorithm and show that the number of distance queries needed is $O(\log n)$ in expectation. Our proof is based on the well-known observation that the nearest neighbor changes $O(\log n)$ times in expectation when traversing the sequence of points, combined with a packing argument certifying that only a constant number of distances needs to be computed in-between two minima. Our experimental results match the theoretical predictions.

5.2 Background

Doubling dimension.

Definition 5.1. *A metric space is called* doubling *with doubling constant k, if every ball of radius r can be covered by at most k balls of radius $r/2$, and k is the smallest number having that property. The doubling dimension of a doubling space is defined as* $\log k$ *(we always use* log *to denote the logarithm with base 2).*

A subspace of a space with doubling dimension d is always of doubling dimension $\leqslant 2d$.

In the remainder of the chapter, we will assume throughout that every considered metric space has a constant doubling dimension.

Well-Separated Pair Decomposition.

Definition 5.2. *Given $t > 1$, two disjoint subsets A, B of a metric space $(\mathcal{M}, \delta)$ are called* t-well-separated*, if*

$$\forall a \in A\ \forall b \in B\ \delta(a,b) \geqslant t \max(\mathrm{diam}(A), \mathrm{diam}(B))$$

A well-separated pair decomposition *(WSPD) is a set of unordered pairs of sets $\{\{A_1, B_1\}, \ldots, \{A_s, B_s\}\}$ such that each pair $\{A_i, B_i\}$ is t-well-separated, and for every unordered pair $\{a, b\}$ of distinct points of $\mathcal{M}$ there exists a unique j such that $a \in A_j$ and $b \in B_j$.*

The notion of WSPD was introduced by Callahan and Kosaraju [33] for Euclidean spaces. Har-Peled and Mendel [93] introduced the notion of net-trees and generalized the results of [33] for WSPD, proving the following:

1. A net-tree for a metric space with n points can be constructed in $2^{O(\dim)} n \log n$ expected time.
2. If $\{\{A_1, B_1\}, \ldots, \{A_s, B_s\}\}$ is an $\varepsilon/16$-WSPD on $\mathcal{M}$, and $a_i \in A_i, b_i \in B_i$ for $i = 1 \ldots s$ are chosen arbitrarily, then we get an ε-spanner by taking s edges (a_i, b_i).
3. For $\varepsilon \in (0, 1]$, an ε-WSPD of size $n\varepsilon^{-O(\dim)}$ can be constructed in $2^{O(\dim)} n \log n + n\varepsilon^{-O(\dim)}$ expected time. The algorithm uses the net-tree structure.

The algorithm of constructing a net-tree is complicated and not easy to implement. Beygelzimer et al. [19] introduced the notion of a cover tree, which is a simpler data structure than net-trees. We mention in passing that cover trees can also be used for building a spanner (this can be proven with the same methods), and we use cover trees for building WSPD spanners in one of our implementations.

Spanners and known constructions. Let $(\mathcal{M}, \delta)$ be a finite metric space with n points. One way to encode the metric space is a complete weighted graph on $\mathcal{M}$, where the weights correspond to the distances of the points. A subgraph G of this graph is called a $(1+\varepsilon)$-*spanner* for $(\mathcal{M}, \delta)$ if for any pair of points (u, v), the shortest path distance d_{uv} of u and v in G satisfies $d_{u,v} \leqslant (1+\varepsilon)\delta(u, v)$. In other words, the shortest path metric of G is a good approximation of the actual distance for every pair of points. Clearly, it is a necessary condition that G is connected, hence every spanner must have at least $n - 1$ edges.

The *greedy spanner* [3] is a simple algorithm to compute linear-sized spanners:

function GreedySpanner(P, ε)
  $E \leftarrow \emptyset$
  Sort all pairwise distances of points in P
  for all pairs (p_i, p_j) in increasing order **do**
    $d_{ij} \leftarrow$ Shortest path distance in (P, E)
    if $d_{ij} > (1+\varepsilon)\delta(p_i, p_j)$ **then**

        Add weighted edge (p_i, p_j, v) to E
    return (P, E)

The greedy spanner is guaranteed [3] to return a spanner of size $O(n)$ (for constant doubling dimension and fixed $\varepsilon > 0$); in an experimental study [62] it was also shown to return the sparsest graph. However, it has to compute all $\binom{n}{2}$ pairwise distances for sorting; this means that in our cost model, the greedy spanner has the worst possible performance.

On the other hand, spanner constructions based on Well-Separated Pair Decomposition [33, 93] only compute $O(n \log n + n\varepsilon^{-d})$ distances to construct a $(1+\varepsilon)$-spannerin doubling dimension d. The spanner size is $O(n\varepsilon^{-d})$. Assuming ε and d again as constants, this construction yields a $O(n)$-size spanner using only $O(n \log n)$ distance computations. However, the algorithm is significantly more involved.

5.3 Blind spanners

We introduce a new framework for constructing spanners which we call *blind spanners*: the idea is to maintain, for every pair of points (p_i, p_j), a lower bound a_{ij} and an upper bound b_{ij} for $\delta(p_i, p_j)$, initially set to 0 and ∞, respectively. While there exists some pair for which $\frac{b_{ij}}{a_{ij}} > (1+\varepsilon)$, we pick one of them, compute its distance and update the lower and upper bounds of all pairs with respect to the newly acquired information. Here is the pseudocode:

function BlindSpanner(P, ε)
    $E \leftarrow \emptyset$
    $a_{i,j} \leftarrow 0$ for all $1 \leqslant i, j \leqslant n$
    $b_{i,j} \leftarrow \infty$ for all $1 \leqslant i, j \leqslant n, i \neq j$
    while $\exists i \neq j : b_{i,j}/a_{i,j} > 1 + \varepsilon$ **do**
        $(i, j) \leftarrow$ GetNextEdgeToAdd()
        $v \leftarrow \delta(p_i, pj)$
        Add weighted edge (p_i, p_j, v) to E
        UpdateBounds(i, j, v)

In this pseudocode we adopt the convention that a positive number divided by 0 is ∞ and ∞ is larger than any real number, thus making the predicate in the while loop well-defined.

We give the details of the UpdateBounds procedure next. Suppose that $\delta(p_i, p_j) = v \in \mathbb{R}$ has been computed. First, we reset $a_{i,j}$, $a_{j,i}$, $b_{i,j}$ and $b_{j,i}$ to v, since the distance of p_i and p_j is exactly v. To update the upper bound of some entry $b_{k,\ell}$, we observe that the shortest path from p_k to p_ℓ might now go through the new edge. Hence, we update

$$b_{k,\ell} \leftarrow \min_{i,j}\{b_{k,\ell}, b_{k,i} + v + b_{j,\ell}, b_{k,j} + v + b_{i,\ell}\}$$

Repeating this for all k, ℓ yields the updated upper bounds. Note that this results in $O(n^2)$ arithmetic operations, but no distance computation.

For the lower bound, we observe that for any $1 \leqslant k, \ell \leqslant n$,

$$v - b_{k,i} - b_{\ell,j}$$

is a lower bound for $\delta(p_k, p_\ell)$. Indeed, this follows from the triangle inequality

$$\delta(p_i, p_j) \leqslant \delta(p_i, p_k) + \delta(p_k, p_\ell) + \delta(p_\ell, p_j)$$

by rearranging terms and plugging in the upper bounds for $\delta(p_i, p_k)$ and $\delta(p_k, p_\ell)$. An analogue bound holds with i and j swapped.

Moreover, the inequalities

$$a_{j,\ell} - v - b_{k,i} \leqslant \delta(p_k, p_\ell)$$
$$a_{j,k} - v - b_{j,i} \leqslant \delta(p_k, p_\ell)$$

hold by triangle inequality, and the same is true with i and j swapped. This yields six lower bounds for $\delta(p_k, p_\ell)$, and $a_{k,\ell}$ is updated to the maximum of these six lower bounds and its current value.

Heuristics. The last missing ingredient of our algorithm is the procedure GetNextEdgeToAdd, that is, how to select the next distance to be computed. We propose two natural choices

BlindRandom Among all pairs (i, j) where $\frac{b_{i,j}}{a_{i,j}} > (1 + \varepsilon)$, we pick one uniformly at random

BlindGreedy Pick the pair (i, j) which maximizes the ratio $\frac{b_{i,j}}{a_{i,j}}$. If the maximizing pair is not unique, choose among the maximizing pairs uniformly at random.

The idea behind BlindGreedy is that we query an edge for which we know the least, in that way hoping to gather most additional information about the metric space. Also, our conventions imply that in BlindGreedy the edges with $a_{i,j} = 0$ or $b_{i,j} = \infty$ have the highest priority, so the algorithm first ensures that the graph is connected and there are positive lower bounds for every edge before it will start adding any other edges. Based on this observation, we also tested variations of the BlindRandom algorithm, where the algorithm first enforces connectedness and/or lower bounds (i.e., if there are infinite upper bounds, then the algorithm can only choose one of the corresponding edges, etc).

The next two heuristics assume the existence of a 2-approximation algorithm for distance computation. Denoting by $A_{i,j}$ the number satisfying $\delta(p_i, p_j) \leqslant A_{i,j} \leqslant 2d(p_i, p_j)$, we sort all pairwise distances according to the values $A_{i,j}$. This yields a roughly sorted sequence of distance, because when $\delta(p_i, p_j) > 2\delta(p_k, p_\ell)$, then $A_{i,j} > A_{k,\ell}$ is guaranteed. We propose two further heuristics that attempt to make use of this sorted sequence.

BlindQuasiSortedGreedy Traverse the pairs in increasing order with respect to $A_{i,j}$.

BlindQuasiSortedShaker Alternates between pairs with small and large $A_{i,j}$ by traversing in increasing order of $A_{i,j}$ in odd iterations and in decreasing order in even iterations.

BlindQuasiSortedGreedy tries to mimic the greedy spanner and hence appears as a natural choice at first sight. However, anticipating the experimental results, the heuristic yields very poor results. The reason is that no pair acquires useful lower bounds when only short distance are queried (the greedy spanner does not have this issue because it knows the distance and hence does not need lower bounds). Generally speaking, short distances are good for sharp upper bounds, whereas long distances are useful for lower bounds. This motivates BlindQuasiSortedShaker which alternates between short and long distances.

5.4 Experiments on spanners

We performed two types of experiments: on points sampled from a low-dimensional Euclidean space and on persistence diagrams with q-Wasserstein distance W_q. Clearly, our model is not meaningful in Euclidean spaces because distance computations are cheap. However, in this case we can easily check the performance of our heuristics on a variety of easily controllable data sets. In order to test the BlindQuasiSorted algorithms, we multiply the true distance by a factor from $[1, 2]$ chosen uniformly. By *sparseness* of a graph with n vertices we always mean the number of edges of the graph divided by $\binom{n}{2}$, the number of edges in the complete graph on n edges.

We tested the algorithms for $\varepsilon \in \{ 0.01, 0.1, 0.2, 0.5 \}$ on the following sets of points in dimensions $d = 2, 3, 4, 5$:

1. In the **uniform** test set points are sampled uniformly at random from the unit cube in $\mathbb{R}^d$.
2. In the **normal** test set points are sampled from the standard normal distribution in $\mathbb{R}^d$.
3. In the **clustered** test set we first sample cluster centers uniformly at random from $[0, 10000]^d$, and then we add normally distributed noise around each of the centers. The number of clusters is chosen so that each cluster contains 50 points.
4. The test set **exp** consists of points of the form $(2^{\xi_1}, \ldots, 2^{\xi_d})$, where ξ_i's are i.i.d. random variables with uniform distribution on $[1, 25]$.

In addition to the standard Euclidean norm on $\mathbb{R}^n$, we used also ℓ_1 and ℓ_∞ norms. In all these experiments, the algorithms that we tested compared in the same way, so we only present results for the **uniform** point set in dimension 2.

Figure 5.2 shows the number of edges of the spanner for various variants of blind and non-blind spanner constructions. Note that for all blind spanner variants, the number of computed distances is equal to the spanner size, while for the non-blind greedy spanner, this number is always $\binom{n}{2}$ and for WPSD it is lower bounded by the size of the spanner. We can see that, even though none of the blind spanners produces spanners of the

same quality as the standard greedy algorithm, BlindGreedy and all variants of BlindRandom perform significantly better than both variants of BlindQuasiSorted.

WSPD spanners performed poorly in our experiments on non-clustered data. We implemented two versions of WSPD: one for the Euclidean case, using quad-trees and the algorithm from [66], and WSPD for general metric spaces with cover trees (using the base $\tau = 1.3$). They both give similar results, and we can only conclude that the advantage of WSPD shows up on larger point sets than the ones we deal with.

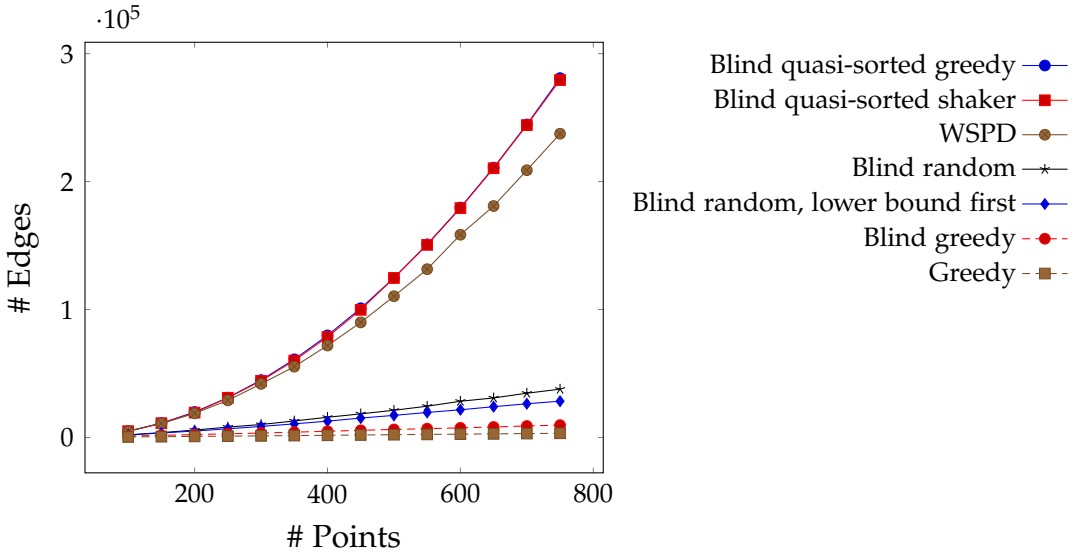


Figure 5.2: Number of edges in blind spanners generated by different variants of the blind algorithm. Greedy non-blind algorithm and WSPD algorithm are included for comparison. The plot is for uniformly distributed points in dimension 2, $\varepsilon = 0.1$.

From our experiments, we conclude that the algorithm with the highest chance of saving a significant amount of distance computations on real data is BlindGreedy. Even though BlindRandom also substantially reduces the number of computed distances, its performance is worse; other algorithms that we tested do not produce sparse spanners. While it would seem plausible that access to approximate value of the distance could be exploited in

	$q=1$		$q=2$		$q=3$	
	Greedy	Bl. greedy	Greedy	Bl. greedy	Greedy	Bl. greedy
$\varepsilon = 0.1$	0.14	0.28	0.32	0.72	0.41	0.78
$\varepsilon = 0.2$	0.06	0.19	0.13	0.44	0.20	0.52
$\varepsilon = 0.5$	0.02	0.10	0.03	0.20	0.05	0.27

Table 5.1: Sparseness of BlindGreedy and Greedy spanners for 450 *original* persistence diagrams with Wasserstein distance W_q.

	$q=1$		$q=2$		$q=3$	
	Br.-force	Bl. greedy	Br.-force	Bl. greedy	Br.-force	Bl. greedy
$\varepsilon = 0.1$	2605	1266	3719	4085	11781	13557
$\varepsilon = 0.2$	2209	878	3338	2574	10541	9383
$\varepsilon = 0.5$	1744	483	2835	1233	9418	5146

Table 5.2: Time required to approximate all pairwise distances with BlindGreedy algorithm, compared to brute-force. Data is for 450 *original* persistence diagrams with Wasserstein distance W_q, time in seconds.

the spanner construction, we could not find a working heuristic; BlindQuasiSortedGreedy and BlindQuasiSortedShaker produce extremely dense spanners. More data from experiments in $\mathbb{R}^n$ can be found in Section 5.5.

The other data sets that we used are the persistence diagrams computed from the McGill shape benchmark [106]. More precisely, we generated two sets of diagrams, *original* and *perturbed*. The *original* diagrams were computed on the points of the 450 original point clouds from the shape data set; we only used persistence in dimension 0, since the diagrams in this dimension are larger in size. For the *perturbed* data set, we added random noise to points of the original shapes, creating 10 modified versions of each of them; then we computed persistence diagrams in dimension 0 of these modified shapes. Persistence diagrams were computed with the DIPHA library [7]. Wasserstein distance were computed approximately with the software library Hera using a relative error of 0.01. We restricted to approximate computation

because computing the Wasserstein distance exactly (using the Hungarian algorithm) has been reported to be very expensive [72].

Sparseness values of BlindGreedy spanner on the whole *original* data set for different values of ε and q are presented in Table 5.1; the Greedy algorithm is used as baseline. We observe that for higher values of q the advantage of our algorithm fades away. A plausible explanation is that for higher values of q, the doubling dimension becomes larger; this is confirmed by the increase in size of the greedy spanner. For W_1 we observe more encouraging figures, avoiding 70 % of distance computations for $\varepsilon = 0.1$; for a 1.5-approximation, we reduce the number of distances computed by a factor of 10.

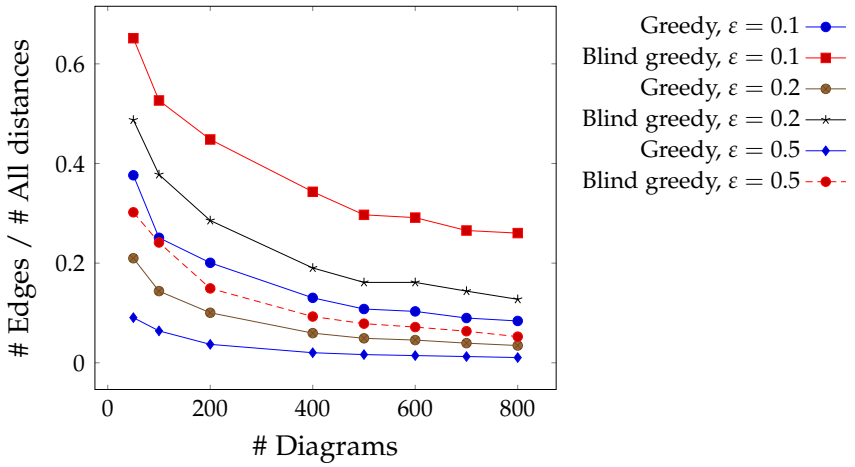


Figure 5.3: Sparseness of BlindGreedy spanner algorithm. The plot is for *perturbed* persistence diagrams, $\varepsilon = 0.1, 0.2, 0.5$ with W_1 distance.

We also compare the time that is needed to compute the matrix of all pairwise distances naively with the time that is needed to approximate it with BlindGreedy spanner. The timings are given in Table 5.2. We compared our approach for a fixed ε with the time to compute all pairwise-distance between the persistence diagrams using Hera, using the same approximation parameter ε. Let us emphasize that the diagrams we have are rather small, with average cardinality 350 points. Nevertheless, on this data set BlindGreedy spanner provides a substantial improvement in terms of real running time, and we expect this advantage to grow for diagrams of larger cardinality.

In order to see the growth of the spanner size, we use the *perturbed* data set, drawing from it random samples. We plot the sparseness of the spanners obtained by Greedy and BlindGreedy algorithms in Figure 5.3 for different values of ε, always using the W_1 distance. As expected, the ratio of computed distances decreases as the number of diagrams grows. In Figure 5.4 we show the growth of the spanner size relative to the number of diagrams. For $\varepsilon = 0.5$ the BlindGreedy spanner starts to stabilize (i.e., demonstrates linear growth) for larger values of n.

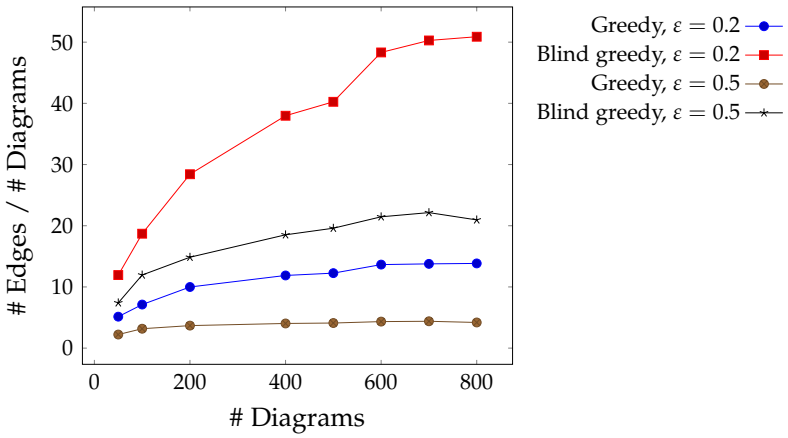


Figure 5.4: Ratio # edges / # diagrams for BlindGreedy spanner. The plot is for 0-dimensional diagrams of perturbed McGill data $\varepsilon = 0.1, 0.2, 0.5$ with W_1 distance.

5.5 Additional Experimental Results on Spanners

In this section we include plots that give more information about the behavior of BlindGreedy spanner in Euclidean space.

Figure 5.5 shows the ratio of the number of edges to the number of points. The ideal behavior is demonstrated by the non-blind greedy spanner, for which this ratio stays practically constant, confirming the linear growth. None of the blind algorithms seems to have this property, but among them the BlindGreedy spanner is the best one. If we assume that the number of edges is proportional to n^α, then we can try to estimate α by

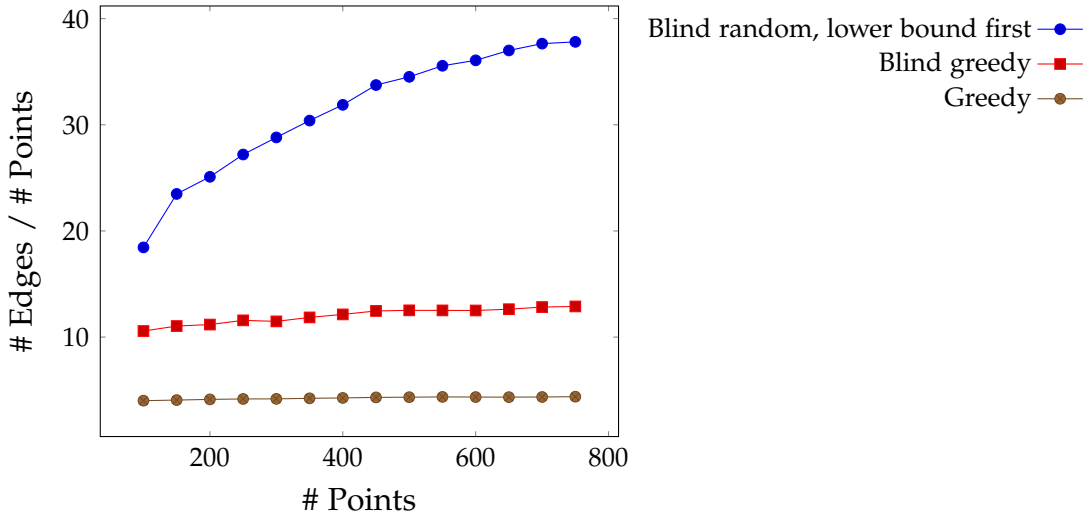


Figure 5.5: Ratio # edges / # points for different variants of spanner algorithms. The plot is for normally distributed points in dimension 2, $\varepsilon = 0.1$.

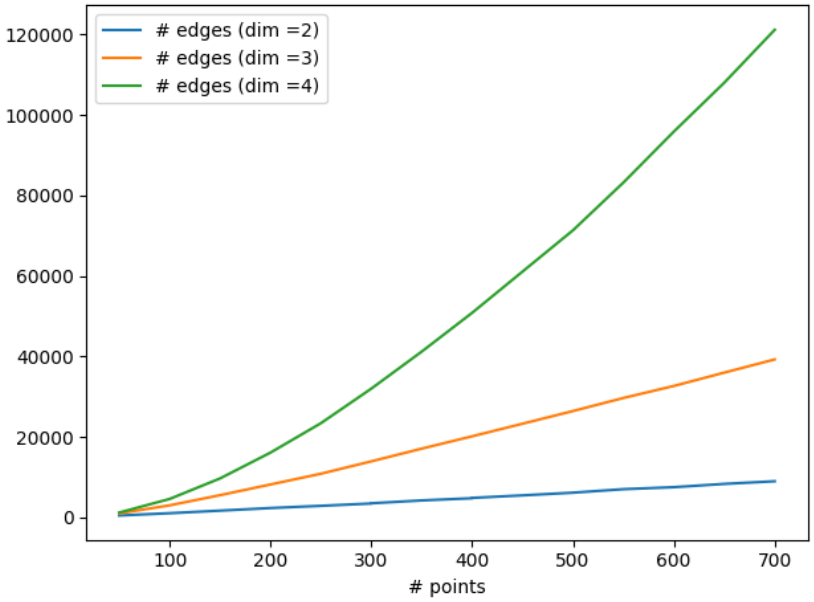


Figure 5.6: Results of BlindGreedy spanner for different dimensions.

linear regression (after taking log). We give in Table 5.3 the estimated exponents α for BlindGreedy and standard greedy algorithms. Note that even for the greedy algorithm these estimated exponents can be significantly larger than 1, which is explained by the fact that the number of points on which we computed spanners is not large enough to clearly see the linear dependence.

The plot in Figure 5.6 compares performance of BlindGreedy in different dimensions. We see that already in dimension 4, it produces a graph with roughly $\frac{1}{2}\binom{n}{2}$ edges for 700 points, which clearly shows some degrading for higher dimensions.

The plot in Figure 5.7 compares the BlindGreedy and Greedy algorithms on **uniform** point sets for different choices of ε. Dependence on ε is approximately the same for both algorithms. Since it is not clearly seen from the picture, we also note that the ratio of the number of edges decreases for smaller values of ε: for $\varepsilon = 2$ the blind greedy spanner contains almost 6 times more edges than the greedy spanner, while for $\varepsilon = 1/32$ the ratio is 2.6

5.6 Approximate nearest neighbors

We consider the standard problem of finding an approximate nearest neighbor: given n points $P = \{ p_1, \ldots, p_n \}$, a query point q and a real number $\varepsilon > 0$, find p_i such that $\delta(p_i, q) \leqslant (1 + \varepsilon) \min_k \delta(q, p_k)$. We also use the shorthand notation

$$r_i := \delta(p_i, q).$$

dimension	**Greedy (non-blind)**	**Blind greedy**
2	1.08	1.12
3	1.24	1.41
4	1.42	1.77

Table 5.3: Estimated exponents in the $|E| = C|V|^{\alpha}$ dependence of the number of edges on the number of points. The data is for $\varepsilon = 0.1$ and for uniform points.

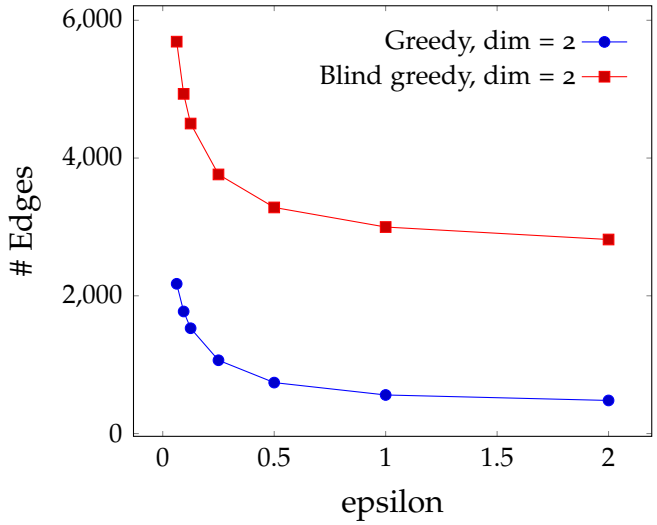


Figure 5.7: Number of edges in the blind greedy and greedy spanners for different values of ε. Data is for 400 normally distributed points in $\mathbb{R}^2$ and $\mathbb{R}^3$.

We assume for simplicity that all exact pairwise distances $\delta(p_i, p_j)$ are already computed (a slight modification of the algorithm can also be applied if only a spanner is available). Our goal is to reduce the number of computed distances $\delta(p_i, q)$ involving the query point q.

Our approach can be summarized as follows. Fix a random permutation of the points of P and consider the points in that order (to simplify notation, we re-index them, so the order is again $p_1, \ldots, p_n$). During the loop, we maintain lower bounds of each p_i to the query point q, which are initially all set to 0. We also remember the closest neighbor c that we have seen so far and its distance v to q. We refer to the point c as the *candidate*. We maintain the invariant that c is an approximate nearest neighbor to q for the points $\{p_1, \ldots, p_i\}$. When reaching the point p_i, we check whether the lower bound a_i satisfies $a_i \geqslant \frac{v}{1+\varepsilon}$. If so, c remains an approximate nearest neighbor and we do not query the distance of p_i to q. Otherwise, we compute $\delta(p_i, q)$ and update the lower bounds of all points according to the newly computed distance. If p_i is closer to q than c, we update c and v accordingly. At the end of the loop, c is an approximate nearest neighbor. The pseudocode follows.

function ApproximateNearestNeighbor(P, q, ε)
  $[p_1, \ldots, p_n] \leftarrow$ random permutation of P
  $a_i \leftarrow 0$ for $i = 1, \ldots, n$ ▷ a_i is lower bound for $\delta(p_i, q)$
  $c \leftarrow p_1, \quad v \leftarrow \delta(p_1, q)$ ▷ c keeps the current candidate
  UpdateBounds(p_1, v)
  for $i = 2 \ldots n$ **do**
    if $a_i \geqslant \frac{v}{1+\varepsilon}$ **then**
      continue
    else
      Compute $r_i = \delta(p_i, q)$
      UpdateBounds(p_i, r_i)
      if $r_i < v$ **then**
        $c \leftarrow p_i, \quad v \leftarrow r_i$
  return c, v

We remark that we obtain an exact nearest neighbor algorithm when setting ε to 0, which means replacing the condition in the if-statement of the loop with $a_i \geqslant v$.

The procedure to maintain the lower bounds a_i is very simple and follows directly from triangle inequality.

procedure UpdateBounds(p_i, r_i)
  for $k = i + 1, \ldots, n$ **do**
    $a_k \leftarrow \max(a_k, |\delta(p_i, p_k) - r_i|)$

Theorem 5.3. *If $(\mathcal{M}, \delta)$ is a doubling space, then, for any fixed $\varepsilon > 0$ the algorithm computes $O(\log n)$ distances $\delta(p_i, q)$ in expectation.*

The full proof is given in Section 5.7, and we only sketch the main ideas here. First, we notice that points "far away" from the current candidate c can be disregarded in the algorithm. More precisely, if p_i is outside of the ball of radius $2\delta(c, q)$ centered at c, the lower bound for p_i will be so large that the distance to of p_i to q does not have to be queried. Second, whenever a distance of q to some p_j is computed, we can disregard all points "very close" to p_j. Precisely, points in a ball of radius $\varepsilon\delta(p_j, q)$ will not invalidate the current candidate to be an approximate nearest neighbor and hence, no distance query for such points is needed.

Now, traversing the points P in random order, we call a point a *minimum* if it is closer to q than all previous points. Note that minima do not necessarily correspond to the candidates in the algorithm because we look for an approximate neareast neighbor only. Standard backwards analysis [95] shows that the expected number of minima encountered is $O(\log n)$. The first two ideas together with a packing argument show that between two consecutive minima, the algorithm queries only a constant number of distances, and the bound follows.

The proof strategy fails for $\varepsilon = 0$ because the ε-ball around p_j as described above degenerates, and the packing argument fails. Indeed, as the example in Figure 5.9 (in Section 5.7) shows, there are point sets where the expected number of distance computations for exact nearest neighbor is linear.

Finally, a fast 2-approximation algorithm for δ leads to a straight-forward optimization: compute a 2-approximation of $\delta(p_i, q)$ for all $1 \leqslant i \leqslant n$ and let m denote the minimal approximate distance encountered. Then, we can discard all points whose approximate distance is larger than $2m$, and run the above algorithm on the remaining points.

5.7 Proof of theorem 5.3

The following lemma is just a reformulation of the well-known packing lemma for doubling spaces (see [98], Sect. 2.2).

Lemma 5.4. *Let $(\mathcal{M}, \delta)$ be a metric space of doubling dimension d, and let P be a subset of a ball $B(x, R)$ in $\mathcal{M}$ such that the distance between any two distinct points of P is at least r. Then*

$$|P| \leqslant \left(\frac{4R}{r}\right)^d$$

Proof. We can cover $B(x, R)$ with 2^d ball of radius $R/2$, each of these balls we can cover with 2^d balls of radius $R/4$, etc. Repeating this process $m :=$

$\lceil \log \frac{R}{r/2} \rceil$ times, we cover $B(x, R)$ with 2^{md} balls of radius at most $r/2$. Since a ball of radius $r/2$ can contain at most one point from P,

$$|P| \leqslant 2^{md} = 2^{\lceil \log \frac{R}{r/2} \rceil d} \leqslant 2^{(1+\log \frac{R}{r/2})d} = \left(\frac{4R}{r}\right)^d. \qquad \square$$

Lemma 5.5. *Assume $r_i = \delta(p_i, q)$ is computed in the algorithm, and let $j > i$.*

1. *If $\delta(p_i, p_j) \geqslant (1 + \frac{1}{1+\varepsilon})r_i$, the algorithm will not compute the distance of p_j to q.*
2. *If $\delta(p_i, p_j) \leqslant \frac{\varepsilon}{1+\varepsilon} r_i$, the algorithm will not compute the distance of p_j to q.*

Lemma 5.5 can be summarized as follows: if $\delta(p_i, q)$ is computed in the algorithm, further distance computations of points very close to p_i or very far from p_i will be avoided.

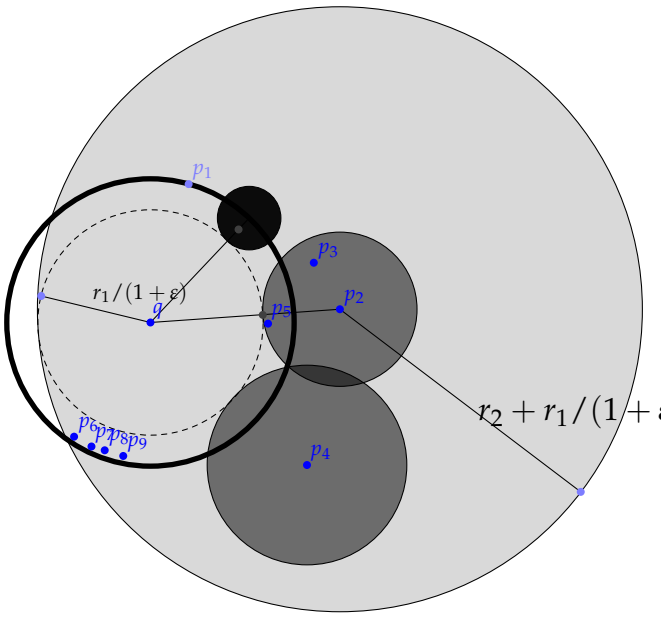


Figure 5.8: First two steps of the ANN algorithm. The small black ball between the dashed circle and the solid circle has radius $v_1\varepsilon/(1+\varepsilon)$; it is the ball that we use in the packing argument, because it is smaller than any of the lightly shaded balls that correspond to points like p_2 and p_4, that is, the points that do not improve v.

Proof. The algorithm computes r_i by assumption and updates all lower bounds. For p_j, it sets $a_j \leftarrow \max(a_j, |\delta(p_i, p_j) - r_i|)$. If $\delta(p_i, p_j) \geqslant (1 + \frac{1}{1+\varepsilon})r_i$, it follows that

$$a_j \geqslant (1 + \frac{1}{1+\varepsilon})r_i - r_i = \frac{r_i}{1+\varepsilon}.$$

Likewise, if $\delta(p_i, p_j) \leqslant \frac{\varepsilon}{1+\varepsilon}r_i$,

$$a_j \geqslant r_i - \delta(p_i, p_j) \geqslant r_i - \frac{\varepsilon}{1+\varepsilon}r_i = \frac{r_i}{1+\varepsilon}.$$

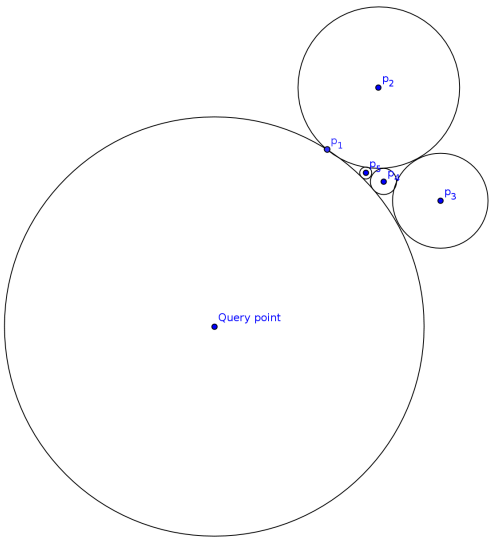


Figure 5.9: Example of a point set where exact nearest neighbor search cannot be accelerated by maintaining bounds. The exact nearest neighbor is the point p_1, next point p_i is placed in the curvilinear triangle formed by the balls around the query point, p_2 and p_{i-1}. Even verifying that p_1 is the true nearest neighbor cannot be done without computing all distances $\delta(p_i, q)$. Indeed, every computed $\delta(p_i, q)$ allows to exclude the region in the corresponding ball around p_i, but all these balls contain only one p_i.

In both cases, after the point p_i is handled, $v \leqslant r_i$ clearly holds. Since v is only decreasing and a_j is only increasing in the algorithm, it follows that $a_j \geqslant \frac{v}{1+\varepsilon}$ when p_j is handled, so the algorithm proceeds without a distance computation. □

In Figure 5.8 we illustrate this. First, p_1 is chosen as the current candidate, and we must compute $\delta(p_2, q)$. After that the algorithm will not compute distance to any of the points inside the heavily shaded ball or outside the lightly shaded ball that are centered at p_2, because their lower bounds allow us to discard them.

In what follows, we let c_i denote the candidate at the end of the i-th iteration of the loop, and v_i the distance to $\delta(c_i, q)$, $i = 1, \ldots, n$. Clearly, $v_1, \ldots, v_n$ is a decreasing sequence. With the previous lemma, we can derive an upper bound for the number of distance computations in an arbitrary subsequence of $p_1, \ldots, p_n$ as follows.

Lemma 5.6. *Among the points $p_k, \ldots, p_\ell$ with $1 \leqslant k < \ell \leqslant n$, the algorithm computes at most*

$$\left(\frac{4(2+\varepsilon)v_k}{\varepsilon v_\ell}\right)^d$$

distances to q.

Proof. By the first part of Lemma 5.5, every point in $p_k, \ldots, p_\ell$ whose distance to q is queried lies in the ball of radius $(1 + \frac{1}{1+\varepsilon})v_k = \frac{2+\varepsilon}{1+\varepsilon}v_k$ around c_k. Moreover, if the distance of two points p_i and p_j with $k \leqslant i < j \leqslant \ell$ is computed, the second part of Lemma 5.5 implies that $\delta(p_i, p_j) > \frac{\varepsilon}{1+\varepsilon}r_i \geqslant \frac{\varepsilon}{1+\varepsilon}v_\ell$. Hence, all points in $p_k, \ldots, p_\ell$ for which the algorithm computes the distance have a pairwise distance of at least $\frac{\varepsilon}{1+\varepsilon}v_\ell$. The statement follows by applying Lemma 5.4. See Figure 5.8. □

A consequence of the lemma is that as long as a candidate c is fixed in the algorithm, the number of computed distances is a constant (since $v_k = v_\ell$). This means that to prove theorem 5.3, it would suffice to show that the candidate changes only a logarithmic number of times in expectation. Let us consider Figure 5.8 again. Notice that the point p_5, which is closer to q than p_1, also will not be a candidate, and at least one of the points p_6, p_7, p_8, p_9

in the annulus between the dashed and solid circle, which are farther from q than p_5, will be chosen as c. This shows that in our algorithm the distance from the candidate to q can drop *slower* than in the brute-force algorithm, thus theorem 5.3 does not immediately follow from standard backwards analysis, and we need to modify it slightly.

Proof. (of theorem 5.3) In the sequence $p_1, \ldots, p_n$, let p_k be a point such that $r_i < r_k$ for all $1 \leqslant i \leqslant k-1$. We call an element of this form a *minimum* of the sequence. A standard backwards analysis argument [95] shows that the probability of p_k being a minimum is at most $1/k$, so that the number of minima in the sequence is $O(\log n)$ in expectation.

Note that for $\varepsilon > 0$, a minimum p_k is not necessarily the candidate c_k because a previous point in the sequence close to p_k might have caused the lower bound a_k to be in the interval $[\frac{v_k}{1+\varepsilon}, v_k]$, which leads to not computing the distance r_k. However, it is true that $v_k \leqslant (1+\varepsilon)r_k$, because otherwise, c_k would not be an approximate nearest neighbor of $\{p_1, \ldots, p_k\}$.

Now, let p_k, p_ℓ be two consecutive minima in the sequence (we also allow that $\ell = n+1$ if k is the last minimum in the sequence). Note that $v_{\ell-1} \geqslant r_k$ because each v_j is equal to r_i for some $i \leqslant j$, and in the sequence $r_1, \ldots, r_{\ell-1}$, r_k is minimal by construction. Using Lemma 5.6, the number of distance computations among the points $p_k, \ldots, p_{\ell-1}$ is at most

$$\left(\frac{4(2+\varepsilon)v_k}{\varepsilon v_{\ell-1}}\right)^d \leqslant \left(\frac{4(2+\varepsilon)(1+\varepsilon)r_k}{\varepsilon r_k}\right)^d = \left(\frac{4(2+\varepsilon)(1+\varepsilon)}{\varepsilon}\right)^d,$$

which is a constant depending only of ε and d, irrespective of the length of the sequence. Since $p_1, \ldots, p_n$ decomposes into $O(\log n)$ such sequences in expectation, the result follows. □

5.8 Experiments on approximate nearest neighbors

In order to experimentally evaluate the performance of our algorithm, we generate random point sets and random query points, and for each query

point run the algorithm 10 times. The average number of distances to the query point that were actually computed is the measure that we are interested in. We average the results over 10 different instances of the point set and query point in order to see the trend clearer; thus each point on the plots in this section is the result of averaging of 100 runs of the code (10 instances, 10 random permutations per instance).

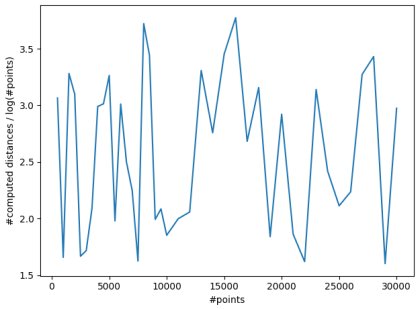


Figure 5.10: Ratio computed distances/ $\log(n)$ for ANN algorithm. Data is for uniformly distributed points.

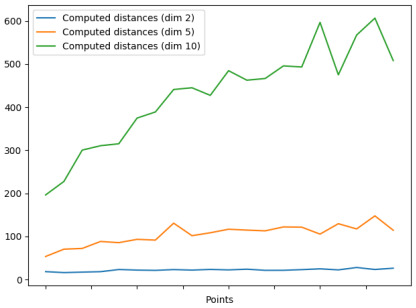


Figure 5.11: Number of computed distances for different dimensions. Points are chosen uniformly, $\varepsilon = 0.01$.

We used the following methods of generating random points:

1. Uniform. Points are sampled uniformly at random from the unit cube in $\mathbb{R}^d$.
2. Normal. Points are sampled from the normal distribution.

Query points were sampled from the uniform distribution on the cube $[-10, 10]^d$ and from the normal distribution centered at the origin with scale 100, thus we get query points that are "inside" the point set and also "outside". We sample data in dimensions up to 20 and for $\varepsilon \in \{ 0.001, 0.005, 0.01, 0.05, 0.1 \}$, the maximal number of points is 30,000.

In order to empirically verify the upper bound $O(\log n)$, we plot the number of computed distances divided by the logarithm of the number of points in Figure 5.10 (for $d = 2$). We see that this ratio, though fluctuating a lot, remains in the interval $[1, 4]$. This not just confirms the theoretical upper bound, but also shows that the algorithm in the low-dimensional

case really computes only a very small number of distances to the query point. As expected, in high dimensions the algorithm does not perform as well. In Figure 5.11 we plot the average number of computed distances for $d = 2, 5, 10$. While for $d = 2$ the growth is hardly noticeable, for $d = 10$ the sublinearity of the growth becomes clear only when the number of points is relatively large, approaching 30000.

Similar experiments were performed on *perturbed* data set of persistence diagrams, which was described in Section 5.4. For a given number n, we randomly select n diagrams from the data set and from the remaining diagrams we randomly choose 20 query points. In Figure 5.12 we plot the average ratio of computed distances to $\log(n)$ for W_1 and W_2 metrics. The average ratio appears to fluctuate around 7 for W_1 and around 8 for W_2, which even makes the algorithm reasonable for practical application, if pairwise distances are already available.

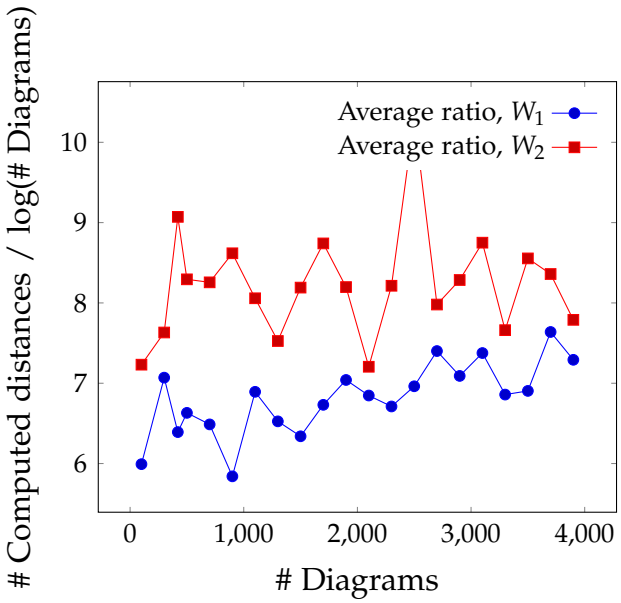


Figure 5.12: Ratio # computed distances/n for ANN algorithm. The plot is for *perturbed* persistence diagrams, $\varepsilon = 0.01$ with W_1 and W_2 metrics.

5.9 Conclusion and future work

We have introduced a new cost model for the analysis of algorithms for metric spaces that fits the situation that computing an individual distance is more costly than other types of primitive operations. Our theoretical results assume that the metric space has a low doubling dimension. However, in our motivating example of collections of persistence diagrams or Reeb graphs, this assumption does not hold. For instance, the space of persistence diagrams has an infinite doubling dimension; see also Section 5.10. Nevertheless, realistic data sets are usually not just a random sample in that infinite-dimensional space, but have structures (e.g. clusters of close-by diagrams) which should be favorable for our approach. Moreover, the size of the BlindGreedy spanner can itself be considered as a piece of information about the dimensionality of the data set under consideration.

In particular, our experiments on real persistence diagrams show that the blind spanner can significantly accelerate computation of all pairwise distances. We must admit that, while in our theoretical model we completely ignore dependency on the number of points n, the $\Omega(n^3)$ complexity of our algorithms makes it practically non-competitive, when n becomes large. On the other hand, this overhead is relatively easy to estimate in advance, just by knowing n, and make a decision, whether BlindGreedy is worth trying.

On the theoretical side, the obvious next question is whether our strategy for blind spanners yields a linear spanner in expectation. It has been brought to our attention[1] that the size of the blind spanner is upper bounded by the *weight* of the WSPD which is the sum of the cardinalities of all pairs in a WSPD. The weight of a WSPD can be quadratic, but preliminary experimental evaluation on worst-case examples do not show such a quadratic behavior. Therefore, we postpone the theoretical analysis of the spanner construction to an extended version of this article.

The existence of a 2-approximation algorithm did not help us to significantly reduce the number of exact distance computations, although it seems

[1] Yusu Wang, personal communication

obvious that knowing the all approximate distances is useful. We pose the question what heuristic could make more use of this feature.

5.10 Appendix: remark on doubling dimension

We provide examples showing that the maps function$f \mapsto \mathrm{Dgm}(f)$ and function$f \mapsto \mathcal{R}(f)$ can increase the doubling dimension by the largest possible factor.

Let n be an integer. Define a piecewise-linear function $f_n\colon [0,1] \to \mathbb{R}$ as follows: f is linear on $[0,1/2]$ and on $[1/2,1]$, and $f_n(0) = 0$, $f_n(1/2) = n$, $f_n(1) = n-1$. As a finite metric space under sup-norm, the set $\{f_n \mid n \geqslant 2\}$ is isometric to $\mathbb{N}$ ($\|f_i - f_j\|_\infty = |i-j|$), hence this set is 1-dimensional. The 0-dimensional persistence diagram $D_n := \mathrm{Dgm}_0(f_n)$ of f_n (with sublevel filtration) consists of two points: $(0,\infty)$ for the connected component of the left branch, and $(n-1,n)$ for the component of the right branch. It is easy to see that the bottleneck distance between D_i and D_j is $1/2$ for all $i \neq j$. Therefore the doubling dimension of the set $\{D_n \mid n = 1,\dots,k\}$ is $\log k$, the highest possible. Thus, the functions f_i provide an example of our claim for the map function $\mapsto$ its persistence diagram.

It takes a bit more work to construct a similar example for Reeb graphs or merge trees. We will do it for merge trees, using the interleaving distance defined in definition 2.37. First, for a fixed $A > 0$, let g_0 be a piecewise-linear function defined by $g_0(A) = A$ and $g_0(x) = 0$ for $x \in (-\infty, 0] \cup [2A, +\infty)$. Put $g_i(x) = g(x+i)$. For each $k \in \mathbb{N}$, we can find some large enough A such that $\|g_i - g_j\| = |i-j|$ for all $1 \leqslant i,j \leqslant k$. Thus the functions g_i form a 1-dimensional discrete metric space, and they all have the same merge tree: $\mathcal{MT}(g_i) = \mathcal{MT}(g_j)$. Second, given a merge tree T, we can always construct a function f such that $\mathcal{MT}f = T$. See Figure 5.13 for an example (digression: this fact can be used for visualization of a function on some high-dimensional domain by a function defined on a rectangle, see [105]).

Third, for each $a > 0$, it is easy to construct a collection of merge trees T_i such that $D_I(T_i, T_j) = a$, see Figure 5.14.

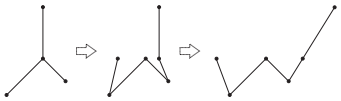


Figure 5.13: Given a merge tree T, it is always possible to construct a function f whose merge tree is T.

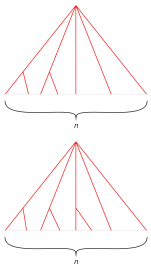


Figure 5.14: Equidistant family of merge trees.

Now, from a collection T_i we can construct the corresponding functions h_i. Appending these functions to the functions g_i, we get the desired family of functions f_i: on the one hand, the shifts of the large peaks in g_i ensure $\|f_i - f_j\| = |i - j|$, on the other hand, $D_I(\mathcal{MT}(f_i), \mathcal{MT}(f_j)) = a$, because attaching the common tree of g_i to the trees of h_i does not change the interleaving distance. An illustration of the construction is shown in Figure 5.15.

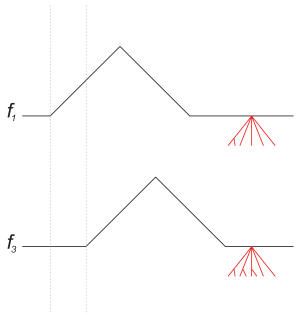


Figure 5.15: Idea of constructing f_i. The left part shows the graphs of g_1, g_3, before the identification process that computes the merge tree. The right part shows the merge trees of h_1, h_3; their graphs have already been collapsed to merge trees.

Bibliography

[1] Pankaj K. Agarwal and R. Sharathkumar. "Approximation algorithms for bipartite matching with metric and geometric costs." In: *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*. 2014, pp. 555–564.

[2] Pankaj K. Agarwal et al. "Computing the Gromov-Hausdorff Distance for Metric Trees." In: *ACM Trans. Algorithms* 14.2 (Apr. 2018), 24:1–24:20. ISSN: 1549-6325.

[3] Ingo Althöfer et al. "On sparse spanners of weighted graphs." In: *Discrete & Computational Geometry* 9.1 (1993), pp. 81–100.

[4] Alexander Andrievsky and Andrei Sobolevskii. *WANN: An Implementation of Weighted Nearest Neighbor Search*. Manual, available at `http://www.mccme.ru/~ansobol/otarie/software.html`. 2008.

[5] Ulrich Bauer, Xiaoyin Ge, and Yusu Wang. "Measuring distance between Reeb graphs." In: *Proceedings of the thirtieth annual symposium on Computational geometry*. ACM. 2014, p. 464.

[6] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. "Clear and compress: Computing persistent homology in chunks." In: *Topological methods in data analysis and visualization III*. Springer, 2014, pp. 103–117.

[7] Ulrich Bauer, Michael Kerber, and Jan Reininghaus. "Distributed computation of persistent homology." In: *2014 proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*. SIAM. 2014, pp. 31–38.

[8] Ulrich Bauer, Claudia Landi, and Facundo Memoli. "The Reeb Graph Edit Distance is Universal." In: *arXiv preprint arXiv:1801.01866* (2018).

[9] Ulrich Bauer and Michael Lesnick. "Induced matchings of barcodes and the algebraic stability of persistence." In: *Proceedings of the thirtieth annual symposium on Computational geometry*. ACM. 2014, p. 355.

[10] Ulrich Bauer and Michael Lesnick. "Persistence Diagrams as Diagrams: A Categorification of the Stability Theorem." In: *arXiv preprint arXiv:1610.10085* (2016).

[11] Ulrich Bauer, Elizabeth Munch, and Yusu Wang. "Strong Equivalence of the Interleaving and Functional Distortion Metrics for Reeb Graphs." In: *31st International Symposium on Computational Geometry (SoCG 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[12] Ulrich Bauer et al. "Phat - Persistent Homology Algorithms Toolbox." In: *J. Symb. Comput.* 78 (2017), pp. 76–90.

[13] Francisco Belchi et al. "Lung topology characteristics in patients with chronic obstructive pulmonary disease." In: *Scientific reports* 8.1 (2018), p. 5341.

[14] Mark de Berg et al. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 978-3540779735.

[15] Dimitri Bertsekas. *A distributed algorithm for the assignment problem*. Tech. rep. Laboratory for Information and Decision Sciences, MIT, 1979.

[16] Dimitri Bertsekas. "The auction algorithm: A distributed relaxation method for the assignment problem." In: *Annals of Operations Research* 14.1 (1988), pp. 105–123.

[17] Dimitri Bertsekas and David Castañon. "Parallel synchronous and asynchronous implementations of the auction algorithm." In: *Parallel Computing* 17.6 (1991), pp. 707–732.

[18] Dimitri Bertsekas and David Castañon. "The auction algorithm for the transportation problem." English. In: *Annals of Operations Research* 20.1 (1989), pp. 67–96. ISSN: 0254-5330.

[19] Alina Beygelzimer, Sham Kakade, and John Langford. "Cover Trees for Nearest Neighbor." In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. Pittsburgh, Pennsylvania, USA: ACM, 2006, pp. 97–104. ISBN: 1-59593-383-2.

[20] Silvia Biasotti et al. "A new algorithm for computing the 2-dimensional matching distance between size functions." In: *Pattern Recognition Letters* 32.14 (2011), pp. 1735–1746.

[21] Håvard Bakke Bjerkevik. "Stability of higher-dimensional interval decomposable persistence modules." In: *arXiv preprint arXiv:1609.02086* (2016).

[22] Håvard Bakke Bjerkevik, Magnus Bakke Botnan, and Michael Kerber. "Computing the interleaving distance is NP-hard." In: *arXiv preprint arXiv:1811.09165* (2018).

[23] Håvard Bjerkevik, Magnus Botnan, and Michael Kerber. "Computing the interleaving distance is NP-hard." arXiv:1811.09165.

[24] Jean-Daniel Boissonnat, Tamal K Dey, and Clément Maria. "The compressed annotation matrix: An efficient data structure for computing persistent cohomology." In: *European Symposium on Algorithms*. Springer. 2013, pp. 695–706.

[25] Magnus Bakke Botnan and William Crawley-Boevey. "Decomposition of persistence modules." In: *arXiv preprint arXiv:1811.08946* (2018).

[26] Magnus Botnan and Michael Lesnick. "Algebraic stability of zigzag persistence modules." In: *Algebraic & geometric topology* 18.6 (2018), pp. 3133–3204.

[27] Alexander M Bronstein, Michael M Bronstein, and Ron Kimmel. "Efficient computation of isometry-invariant distances between surfaces." In: *SIAM Journal on Scientific Computing* 28.5 (2006), pp. 1812–1836.

[28] Peter Bubenik. "Statistical topological data analysis using persistence landscapes." In: *The Journal of Machine Learning Research* 16.1 (2015), pp. 77–102.

[29] Peter Bubenik and Jonathan A Scott. "Categorification of persistent homology." In: *Discrete & Computational Geometry* 51.3 (2014), pp. 600–627.

[30] Peter Bubenik and Tane Vergili. "Topological spaces of persistence modules and their properties." In: *Journal of Applied and Computational Topology* 2.3-4 (2018), pp. 233–269.

[31] Rainer E. Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems, Revised Reprint:* Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2009. ISBN: 9781611972238.

[32] Paul B. Callahan and S. Rao Kosaraju. "A Decomposition of Multidimensional Point Sets with Applications to K-nearest-neighbors and N-body Potential Fields." In: *J. ACM* 42.1 (Jan. 1995), pp. 67–90. ISSN: 0004-5411.

[33] Paul B Callahan and S Rao Kosaraju. "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields." In: *Journal of the ACM (JACM)* 42.1 (1995), pp. 67–90.

[34] G. Carlsson and A. Zomorodian. "The Theory of Multidimensional Persistence." English. In: *Discrete & Computational Geometry* 42.1 (2009), pp. 71–93. ISSN: 0179-5376.

[35] Gunnar Carlsson and Vin De Silva. "Zigzag persistence." In: *Foundations of computational mathematics* 10.4 (2010), pp. 367–405.

[36] Mathieu Carrière and Ulrich Bauer. "On the Metric Distortion of Embedding Persistence Diagrams into Separable Hilbert Spaces." In: *35th International Symposium on Computational Geometry (SoCG 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[37] Mathieu Carriere and Steve Oudot. "Structure and stability of the one-dimensional mapper." In: *Foundations of Computational Mathematics* 18.6 (2018), pp. 1333–1396.

[38] Mathieu Carrière and Steve Oudot. "Local equivalence and intrinsic metrics between Reeb graphs." In: *arXiv preprint arXiv:1703.02901* (2017).

[39] Frédéric Chazal et al. *The structure and stability of persistence modules*. Springer, 2016.

[40] Chao Chen and Michael Kerber. "An output-sensitive algorithm for persistent homology." In: *Computational Geometry* 46.4 (2013), pp. 435–447.

[41] Chao Chen and Michael Kerber. "Persistent homology computation with a twist." In: *Proceedings 27th European Workshop on Computational Geometry*. Vol. 11. 2011, pp. 197–200.

[42] L. Paul Chew and Klara Kedem. "Improvements on Geometric Pattern Matching Problems." In: *Algorithm Theory - SWAT '92, Third Scandinavian Workshop on Algorithm Theory, Helsinki, Finland, July 8-10, 1992, Proceedings*. 1992, pp. 318–325.

[43] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. "Extending persistence using Poincaré and Lefschetz duality." In: *Foundations of Computational Mathematics* 9.1 (2009), pp. 79–103.

[44] David Cohen-Steiner et al. "Lipschitz functions have L p-stable persistence." In: *Foundations of computational mathematics* 10.2 (2010), pp. 127–139.

[45] Justin Curry. "Sheaves, cosheaves and applications." In: *arXiv preprint arXiv:1303.3255* (2013).

[46] Justin Curry, Robert Ghrist, and Vidit Nanda. "Discrete Morse theory for computing cellular sheaf cohomology." In: *Foundations of Computational Mathematics* 16.4 (2016), pp. 875–897.

[47] Vin De Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. "Dualities in persistent (co) homology." In: *Inverse Problems* 27.12 (2011), p. 124003.

[48] Vin De Silva, Elizabeth Munch, and Amit Patel. "Categorified reeb graphs." In: *Discrete & Computational Geometry* 55.4 (2016), pp. 854–906.

[49] Tamal K Dey, Facundo Mémoli, and Yusu Wang. "Multiscale mapper: Topological summarization via codomain covers." In: *Proceedings of the twenty-seventh annual acm-siam symposium on discrete algorithms*. SIAM. 2016, pp. 997–1013.

[50] Tamal K Dey, Facundo Mémoli, and Yusu Wang. "Topological Analysis of Nerves, Reeb Spaces, Mappers, and Multiscale Mappers." In: *33rd International Symposium on Computational Geometry (SoCG 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[51] Tamal K Dey and Cheng Xin. "Computing Bottleneck Distance for 2-D Interval Decomposable Modules." In: *34th International Symposium on Computational Geometry (SoCG 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[52] Tamal Dey and Cheng Xin. "Generalized Persistence Algorithm for Decomposing Multi-parameter Persistence Modules." arXiv:1904.03766.

[53] Barbara Di Fabio and Claudia Landi. "Reeb Graphs of Piecewise Linear Functions." In: *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer. 2017, pp. 23–35.

[54] H. Edelsbrunner and J. Harer. *Computational Topology. An Introduction.* American Mathematical Society, 2010. ISBN: 0-8218-4925-5.

[55] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. "Topological persistence and simplification." In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE. 2000, pp. 454–463.

[56] Herbert Edelsbrunner and Georg Osang. "The Multi-cover Persistence of Euclidean Balls." In: *34th International Symposium on Computational Geometry, SoCG 2018, June 11-14, 2018, Budapest, Hungary*. 2018, 34:1–34:14.

[57] Herbert Edelsbrunner, Ziga Virk, and Hubert Wagner. "Topological Data Analysis in Information Space." In: *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA*. 2019, 31:1–31:14.

[58] Herbert Edelsbrunner and Hubert Wagner. "Topological Data Analysis with Bregman Divergences." In: *33rd International Symposium on Computational Geometry (SoCG 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[59] Jack Edmonds. "Paths, Trees, and Flowers." In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[60] Alon Efrat, Alon Itai, and Matthew J. Katz. "Geometry Helps in Bottleneck Matching and Related Problems." In: *Algorithmica* 31.1 (2001), pp. 1–28.

[61] Elena Farahbakhsh Touli and Yusu Wang. "FPT-algorithms for computing Gromov-Hausdorff and interleaving distances between trees." In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[62] Mohammad Farshi and Joachim Gudmundsson. "Experimental study of geometric t-spanners." In: *Journal of Experimental Algorithmics (JEA)* 14 (2009), p. 3.

[63] Massimo Ferri. "Persistent topology for natural data analysis—A survey." In: *Towards Integrative Machine Learning and Knowledge Extraction*. Springer, 2017, pp. 117–133.

[64] Ellen Gasparovic et al. "Intrinsic Interleaving Distance for Merge Trees." In: *arXiv preprint arXiv:1908.00063* (2019).

[65] Robert W Ghrist. *Elementary applied topology*. Vol. 1. Createspace Seattle, 2014.

[66] Sariel Har-Peled. *Geometric approximation algorithms*. 173. American Mathematical Soc., 2011.

[67] Masaki Hilaga et al. "Topology matching for fully automatic similarity estimation of 3D shapes." In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 203–212.

[68] John E. Hopcroft and Richard M. Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs." In: *SIAM Journal on Computing* 2.4 (1973), pp. 225–231.

[69] Jon L. Bentley. "Multidimensional binary search trees used for associative searching." In: *Communications of the ACM* 18 (1975), pp. 509–517.

[70] Tomasz Kaczynski, Konstantin Mischaikow, and Marian Mrozek. *Computational homology*. Vol. 157. Springer Science & Business Media, 2006.

[71] Tomasz Kaczynski and Marian Mrozek. "The cubical cohomology ring: an algorithmic approach." In: *Foundations of Computational Mathematics* 13.5 (2013), pp. 789–818.

[72] M. Kerber, D. Morozov, and A. Nigmetov. "Geometry Helps to Compare Persistence Diagrams." In: *Journal of Experimental Algorithms* 22 (Sept. 2017), 1.4:1–1.4:20. ISSN: 1084-6654.

[73] Michael Kerber, Michael Lesnick, and Steve Oudot. "Exact computation of the matching distance on 2-parameter persistence modules." In: *35th International Symposium on Computational Geometry (SoCG 2019)*. 2019, 46:1–46:15.

[74] Michael Kerber, Dmitriy Morozov, and Arnur Nigmetov. "Geometry Helps to Compare Persistence Diagrams." In: *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2016, pp. 103–112.

[75] Michael Kerber and Arnur Nigmetov. "Metric Spaces with Expensive Distances." In: *arXiv preprint arXiv:1901.08805* (2019).

[76] Robert Krauthgamer and James R Lee. "The black-box complexity of nearest-neighbor search." In: *Theoretical Computer Science* 348.2-3 (2005), pp. 262–276.

[77] Claudia Landi. "The rank invariant stability via interleavings." In: *arXiv preprint arXiv:1412.3374* (2014).

[78] Michael Lesnick. "The theory of the interleaving distance on multidimensional persistence modules." In: *Foundations of Computational Mathematics* 15.3 (2015), pp. 613–650.

[79] Michael Lesnick and Matthew Wright. "Computing Minimal Presentations and Bigraded Betti Numbers of 2-Parameter Persistent Homology." arXiv:1902.05708.

[80] Konstantin Mischaikow and Vidit Nanda. "Morse theory for filtrations and efficient computation of persistent homology." In: *Discrete & Computational Geometry* 50.2 (2013), pp. 330–353.

[81] Dmitriy Morozov. *Dionysus library for computing persistent homology*. `mrzv.org/software/dionysus`. 2010.

[82] Dmitriy Morozov. "Persistence algorithm takes cubic time in worst case." In: *BioGeometry News, Dept. Comput. Sci., Duke Univ* 2 (2005).

[83] Dmitriy Morozov, Kenes Beketayev, and Gunther Weber. "Interleaving distance between merge trees." In: *Discrete and Computational Geometry* 49.22-45 (2013), p. 52.

[84] David M. Mount and Sunil Arya. *ANN: A Library for Approximate Nearest Neighbor Searching*. `http://www.cs.umd.edu/~mount/ANN`. 2010.

[85] James Munkres. "Algorithms for the Assignment and Transportation Problems." In: *Journal of the Society of Industrial and Applied Mathematics* 5.1 (Mar. 1957), pp. 32–38.

[86] N. Milosavljevic, D. Morozov, and P. Skraba. "Zigzag persistent homology in matrix multiplication time." In: *ACM Symposium on Computational Geometry (SoCG)*. 2011, pp. 216–225.

[87] Salman Parsa. "A Deterministic $O(m \log m)$ Time Algorithm for the Reeb Graph." In: *Discrete & Computational Geometry* 49.4 (2013), pp. 864–878.

[88] Iosif Polterovich, Leonid Polterovich, and Vukašin Stojisavljević. "Persistence barcodes and Laplace eigenfunctions on surfaces." In: *Geometriae Dedicata* 201.1 (2019), pp. 111–138.

[89] Leonid Polterovich and Egor Shelukhin. "Autonomous Hamiltonian flows, Hofer's geometry and persistence modules." In: *Selecta Mathematica* 22.1 (2016), pp. 227–296.

[90] Leonid Polterovich, Egor Shelukhin, and Vukašin Stojisavljević. "Persistence Modules with Operators in Morse and Floer Theory." In: *Moscow Mathematical Journal* 17.4 (2017), pp. 757–786.

[91] Michael Robinson et al. "Geometry and topology of the space of sonar target echos." In: *The Journal of the Acoustical Society of America* 143.3 (2018), pp. 1630–1645.

[92] Ana Romero et al. "Defining and computing persistent Z-homology in the general case." In: *arXiv preprint arXiv:1403.7086* (2014).

[93] S. Har-Peled and M. Mendel. "Fast Construction of Nets in Low Dimensional Metrics and Their Applications." In: *SIAM Journal on Computing* 35 (2006), pp. 1148–1184.

[94] S. Oudot. *Persistence theory: From Quiver Representation to Data Analysis*. Vol. 209. Mathematical Surveys and Monographs. American Mathematical Society, 2015.

[95] Raimund Seidel. "Backwards Analysis of Randomized Geometric Algorithms." In: *New Trends in Discrete and Computational Geometry*. Ed. by Janos Pach. Springer, 1993.

[96] Jean-Pierre Serre. "Homologie singulière des espaces fibrés." In: *Ann. of Math* 54.2 (1951), pp. 425–505.

[97] Gurjeet Singh, Facundo Mémoli, and Gunnar E Carlsson. "Topological methods for the analysis of high dimensional data sets and 3d object recognition." In: *SPBG*. 2007, pp. 91–100.

[98] Michiel Smid. "Efficient Algorithms." In: ed. by Susanne Albers, Helmut Alt, and Stefan Näher. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. The Weak Gap Property in Metric Spaces of Bounded Doubling Dimension, pp. 275–289. ISBN: 978-3-642-03455-8.

[99] Jian Sun, Maks Ovsjanikov, and Leonidas Guibas. "A Concise and Provably Informative Multi-scale Signature Based on Heat Diffusion." In: *Proceedings of the Symposium on Geometry Processing*. SGP '09. Berlin, Germany: Eurographics Association, 2009, pp. 1383–1392.

[100] The RIVET Developers. *RIVET*. Version 1.0. 2018. URL: `http://rivet.online`.

[101] Chris Tralie. *pyhks*. `https://github.com/ctralie/pyhks`. 2018.

[102] Pravin M. Vaidya. "Geometry Helps in Matching." In: *SIAM J. Comput.* 18.6 (1989), pp. 1201–1225.

[103] Vinay Venkataraman, Karthikeyan Natesan Ramamurthy, and Pavan Turaga. "Persistent homology of attractors for action recognition." In: *2016 IEEE international conference on image processing (ICIP)*. IEEE. 2016, pp. 4150–4154.

[104] Hubert Wagner, Chao Chen, and Erald Vuçini. "Efficient computation of persistent homology for cubical data." In: *Topological methods in data analysis and visualization II*. Springer, 2012, pp. 91–106.

[105] Gunther Weber, Peer-Timo Bremer, and Valerio Pascucci. "Topological landscapes: A terrain metaphor for scientific data." In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (2007), pp. 1416–1423.

[106] Juan Zhang et al. "Retrieving articulated 3-d models using medial surfaces and their graph spectra." In: *International workshop on energy minimization methods in computer vision and pattern recognition*. Springer. 2005, pp. 285–300.

[107] Afra Zomorodian and Gunnar Carlsson. "Computing persistent homology." In: *Discrete & Computational Geometry* 33.2 (2005), pp. 249–274.