Geometry Helps to Compare Persistence Diagrams*

Michael Kerber† Dmitriy Morozov‡ Arnur Nigmetov§

Abstract

Exploiting geometric structure to improve the asymptotic complexity of discrete assignment problems is a well-studied subject. In contrast, the practical advantages of using geometry for such problems have not been explored. We implement geometric variants of the Hopcroft–Karp algorithm for bottleneck matching (based on previous work by Efrat el al.), and of the auction algorithm by Bertsekas for Wasserstein distance computation. Both implementations use k-d trees to replace a linear scan with a geometric proximity query. Our interest in this problem stems from the desire to compute distances between persistence diagrams, a problem that comes up frequently in topological data analysis. We show that our geometric matching algorithms lead to a substantial performance gain, both in running time and in memory consumption, over their purely combinatorial counterparts. Moreover, our implementation significantly outperforms the only other implementation available for comparing persistence diagrams.

1 Introduction

The *assignment problem* is among the most famous problems in combinatorial optimization. Given a weighted bipartite graph G with $(n + n)$ vertices, it asks for a perfect matching with minimal cost. A common cost function is the minimum of the sum of the q-th powers of weights of the matching edges, for some $q \geq 1$. We call the solution in this case the *q-th Wasserstein matching* and its cost the *q-th Wasserstein distance*. As q tends to infinity, the Wasserstein distance approaches the *bottleneck distance*, by definition the minimum of the maximum edge weight over all perfect matchings. See [8] for a contemporary discussion of the topic with links to applications.

We consider the geometric version of the assignment problem, where the vertices of G are points in a metric space (X, d), and edge weights are determined by the distance function d. The 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. [16] for the bottleneck distance and Vaidya [23] 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 [2] for a recent summary. On the other hand, there has been no rigorous study of whether geometry also helps *in practice*. Our paper is devoted to this question.

We restrict attention to one scenario that motivates our study of the assignment problem. In the field of *topological data analysis*, the homological information of a data set is often summarized in a *persistence diagram*. Such diagrams, themselves point sets in $\mathbb{R}^2$, capture connectivity of a data set, and, specifically, how the connectivity changes across various scales [14]. Persistence diagrams are stable: small changes in the data cause only small changes in the diagram [10, 11]. Accordingly, the distances between persistence diagrams have received a lot of attention in applications [1, 18, 17]: where persistence diagrams serve as topological proxies for the input data, distances between the diagrams serve as proxy measures of the similarity between data sets. These distances, in turn, can be expressed as a Wasserstein or a bottleneck distance between two planar point sets, using L_∞ as the metric in the plane (see Section 2 for the precise definition and the reduction).

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.

*© SIAM 2016. The definitive version appears in Algorithm Engineering and Experiments (ALENEX16).

†Graz Unvirsity of Technology, Graz, Austria (kerber@tugraz.at)

‡Lawrence Berkeley National Lab, Berkeley, CA, USA (dmitriy@mrzv.org)

§Graz Unvirsity of Technology, Graz, Austria (nigmetov@tugraz.at)

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 [20]. The former prototypical implementation is 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.[1] This paper contains the following specific contributions:

- For bottleneck matchings, we follow the approach of Efrat et al. [16]: they augment the classical combinatorial algorithm of Hopcroft and Karp [19] with a geometric data structure to speed up the search for vertices close to query points. We do not follow their asymptotically optimal but complicated approach. We instead use a k-d tree data structure [4] 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.

- For Wasserstein matchings, we implement a geometric variant of the *auction algorithm*, an approximation algorithm by Bertsekas [5]. 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 [3]. We also implement, for comparison, a version of the auction algorithm that does not exploit geometry; it achieves running times close to the geometric variant, but at the expense of quadratic (vs linear) space complexity. Both geometric and non-geometric 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 [22]; see Section 4.

2 Background

Assignment problem. Given a weighted bipartite graph $G = (A \sqcup B, E)$, 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-th 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-th Wasserstein cost* of G is the cost of an optimal matching. If a graph does not have a perfect matching, its cost is infinite.

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

Persistent homology and diagrams. We are concerned with a particular type of assignment problems in this paper. Specifically, we are interested in distances studied by the *theory of persistent homology*, distances that measure topological differences between objects. In a nutshell, persistent homology records connectivity of objects — connected components, tunnels, voids, and higher-dimensional "holes" — across multiple scales. *Persistence diagrams* summarize this information as two-dimensional point sets with multiplicities. A point (x, y) with multiplicity m represents m features that all appear for the first time at scale x and disappear at scale y. Features appear before they disappear, so the points lie above the diagonal $x = y$. The difference $y - x$ is called the *persistence* of a feature. To make persistence diagrams stable, each point (x, x) on the diagonal is counted with infinite multiplicity.

Given two persistence diagrams X and Y, their *bottleneck distance* is defined as

$$W_\infty(X, Y) = \inf_{\eta: X\to Y} \sup_{x\in X} \|x - \eta(x)\|_\infty,$$

where η ranges over all bijections and $\|(x, y)\|_\infty = \max\{|x|, |y|\}$ is the usual L_∞-norm. Similarly, the *q-th Wasserstein distance* is defined as

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

[1] Bottleneck distance: `https://bitbucket.org/grey_narn/geom_bottleneck`, Wasserstein distance: `https://bitbucket.org/grey_narn/geom_matching`

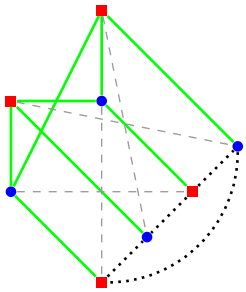


Figure 1: An example of G_q for two persistence diagrams with 2 off-diagonal points each. Skew edges are dashed gray, edges connecting diagonal points are dotted black.

Why are these distances interesting? Because they are stable [10, 11, 13, Ch. VIII.3]: a small perturbation of the measured phenomenon, for example, a scalar function on a manifold, creates only a small change in the persistence diagram — both distances reflect this. The diagonal of a persistence diagram plays a crucial role in stability. Small perturbations may create new topological features, but their persistence is necessarily small, making it possible to match them to the points on the diagonal. We refer the reader to the cited papers for an extensive discussion.

Persistence distance as a matching problem. We assume from now on that persistence diagrams consist of finitely many off-diagonal points with finite multiplicity and all the diagonal points with infinite multiplicity. In this case, the task of computing $W_*(X, Y)$ can be reduced to a bipartite graph matching problem; we follow the notation and argument given in [13, Ch. VIII.4]. Let X_0, Y_0 denote the off-diagonal points of X and Y, respectively. Let X_0' denote the orthogonal projections of X_0 to the diagonal, that is $X_0' = \{((x+y)/2, (x+y)/2) \mid (x, y) \in X_0\}$; this set contains the points on the diagonal that are closest to X_0 in L_∞-distance. With Y_0' defined analogously, we define $U = X_0 \cup Y_0'$ and $V = Y_0 \cup X_0'$; both have the same number of points. For an integer $q > 0$, we define the weighted complete bipartite graph, $G_q = (U \sqcup V, U \times V)$, whose weights are given by the function

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

Points from U and V are depicted as red squares and blue circles, respectively, in Figure 1; all the diagonal points are connected by edges of weight 0 (plotted in dotted black). The following result is stated as the *Reduction lemma* in [13, Ch. VIII.4]:

Lemma 2.1.

- *$W_\infty(X, Y)$ equals the bottleneck cost of G_1.*
- *$W_q(X, Y)$ equals the q-th Wasserstein cost of G_q.*

Consider the weight function c_q. G_q 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's 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 1).

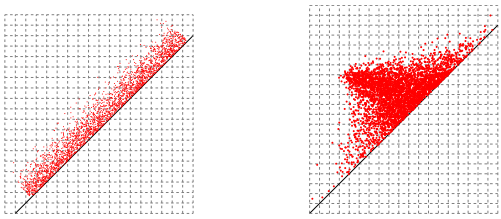
Lemma 2.2. *For both bottleneck and Wasserstein distance, there exists an optimal matching in G_q that does not contain any skew edge.*

Proof. Fix an arbitrary matching M with at least one skew edge. Define the matching M' as follows: For any $uv \in M \cap X_0 \times Y_0$, add uv and $u'v'$ to M', where u' is the projection of u. 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 and a' is its projection. It is easy to see that M' has no skew edges, and its cost is not worse than the cost of M: indeed, the skew edge ab' got replaced by aa' which is strictly smaller, and the vertices on the diagonal possibly got rearranged, which has no effect on the cost. □

Lemma 2.2 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.[2]

K-d trees. K-d trees [4] 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

[2] Dionysus uses the same simplification.



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

Figure 2: Examples of persistence diagrams.

to have roughly the same number of points. Given a query point u, 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 we use the ANN [21] implementation of k-d trees which we change to support deletion of points. For Section 4 we implemented our own version of k-d trees to support search for a nearest neighbor with weights.

Experimental setup. All experiments 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 behavior 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. We refer to this class of experiments as normal instances (Figure 2a).

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[3] 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 2b). 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 have tested various choices of s, the scaling parameter in the normal class, and of the sphere dimension in the real class. We have 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 paper.

3 Bottleneck matchings

Our approach follows closely the work of Efrat et al. [16], 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.

The algorithm by Hopcroft and Karp. Efrat et al. modify the algorithm by Hopcroft and Karp [19] to find a maximum matching. We briefly summarize the Hopcroft–Karp algorithm; [16] 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

[3] http://dipha.googlecode.com

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 [19], 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 $d(q, s) \leq r$. $\mathcal{D}_r(S)$ 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 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 [9], for efficient k-th distance selection for a bi-chromatic point set under the L_∞-distance; see [16, 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 [16, 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 [13, p.196]). Let $G_1 = (U \sqcup V, U \times V)$ be the graph defined in Lemma 2.1. 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 non-empty. The overhead of maintaining S' is negligible. We summarize:

THEOREM 3.1. *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 originally stored in the search tree. We remark that using range trees [12], the worst-case complexity could be further reduced to $O(\log n)$.

Initial tests showed that the naive approach of precomputing 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, $\|u_0 - v_0\|_\infty \leq d_{\max} \leq 3\|u_0 - v_0\|_\infty$ (from the triangle inequality). Setting $t = 3\|u_0 - v_0\|_\infty$,

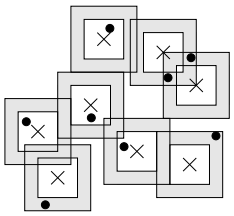


Figure 3: Illustration of the exact computation step: the exact bottleneck distance must be realized by a point in B (circles) in an annulus around A (crosses). The width of the annulus is determined by the approximation quality. In this example, there are 6 candidate pairs.

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, 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 [12]. 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 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 8 for the largest displayed normal instance with 30000 points per diagram. We note 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 4 (left). For the non-geometric version, the best fit appeared for $\alpha = 2.1$, 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.

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 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 [15] 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.

4 Wasserstein matchings

Auction algorithm. The *auction algorithm* of Bertsekas [5] is an asymmetric approach to finding the maximum weight matching. One half of the bipartite graph is treated as "bidders", the second half as "objects." Initially, each object j is assigned zero price,

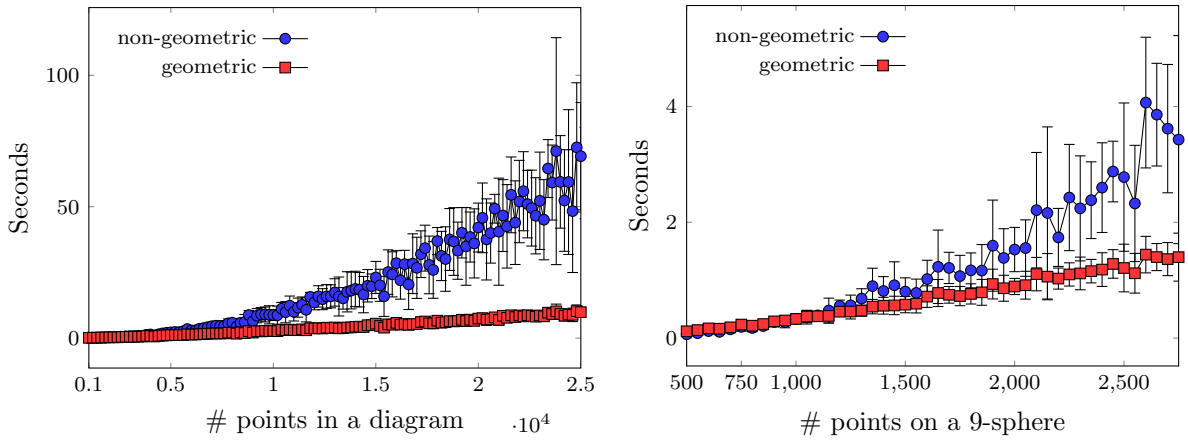


Figure 4: Running times of the bottleneck distance computation on normal data (left) and real data (right) for varying number of points.

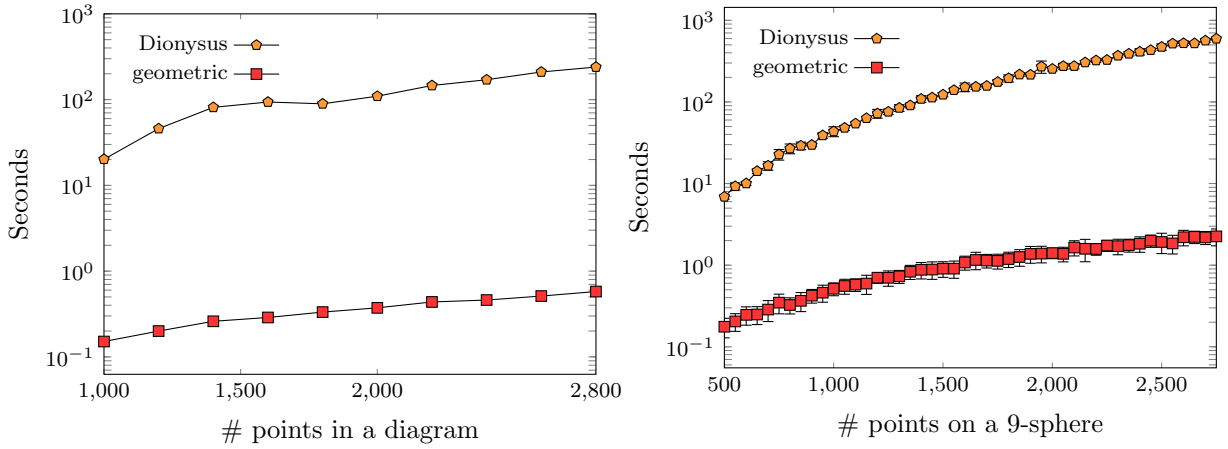


Figure 5: Comparison of our exact geometric bottleneck algorithm with DIONYSUS for normal (left) and real (right) input.

$p_j = 0$, 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 weights in the previous section as the bidder–object benefits, $b_{ij} = -w(i,j) = -d(i,j)^q$. (If the edge (i,j) is not in the graph, $b_{ij} = -\infty$.)

The auction proceeds iteratively. In each iteration, every bidder without an assignment chooses an object with the maximum value, defined as the benefit minus the current price of the object, $v_{ij} = (b_{ij} - p_j)$. Each such bidder is willing to increase the price of the chosen object by an increment, Δp_{ij}, that would make the value of the object equal to the second best choice. When multiple bidders choose the same object j, the one willing to pay the highest increment, $i = \operatorname{argmax}_i \Delta p_{ij}$, wins. The objects are assigned to the winning bidders, who increase their prices by the winning increments. For technical reasons, the changed prices are increased further by some parameter ϵ. The algorithm stops when each bidder is assigned an object.

Small values of ϵ 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* procedure 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 [7], we initialize ϵ with the maximum weight divided by 4 and divide ϵ by 5 when starting a new round.

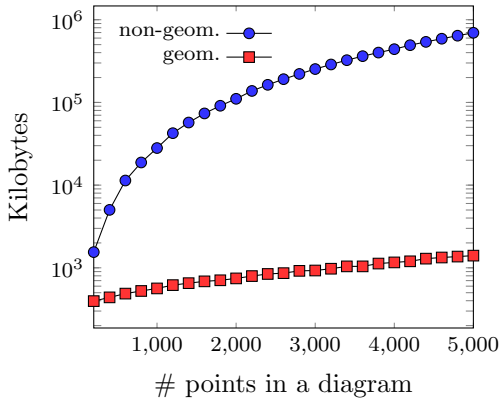


Figure 6: Comparison of memory consumption of geometric and non-geometric versions of auction algorithm on normal instances.

It follows from the properties of the auction algorithm that, if d is the cost of the matching returned by the algorithm and ϵ satisfies $n\epsilon < d^q + (1+\delta)^q d^q$, then d is the δ-approximation of the exact Wasserstein distance o, that is, $d \in [o, (1+\delta)o)$. We use this as a stopping criterion for ϵ-scaling procedure to guarantee the relative error of our result.

Bidding. The computational crux of the algorithm is for a bidder to select the object of maximum value. The brute-force approach is for each bidder to do an exhaustive search over all objects. Doing so requires a quadratic running time per iteration. But let us consider what the search actually entails. Bidder i must find object $j = \operatorname{argmax}_j v_{ij}$. Recall $v_{ij} = b_{ij} - p_j = -d(i,j)^q - p_j$. Maximizing this quantity for a fixed i, over all j, is equivalent to minimizing $d(i,j)^q + p_j$.

The first way to quickly find this answer 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 object 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 object j that minimizes $d(i,j)^q + p_j$ by performing the proximity search in a k-d tree. But, as the prices increase, we need to augment the k-d tree to take them 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 best candidate.

Once a bidder selects the best object, it increases its price. We adjust the subtree weights in the k-d tree by increasing the chosen object's weight and updating the weights on the path to the root accordingly, if the minimal weight has changed. If the minimum does not change at some node in the path, we interrupt the traversal to the root.

The case of persistence diagrams requires some special care. We can distinguish between *diagonal* and *off-diagonal* bidders and objects. Diagonal bidders bid for only one off-diagonal object, according to Lemma 2.2. The k-d tree is only used to determine bids for off-diagonal objects (and, accordingly, store only those). We omit further technical details.

Experiments. Figure 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 6 for diagrams of 5000 points, compared to its combinatorial counterpart. 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 6.

Again, we compare our geometric approach with Dionysus, which uses John Weaver's implementation[4] of the Hungarian algorithm [22]. Figure 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 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

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

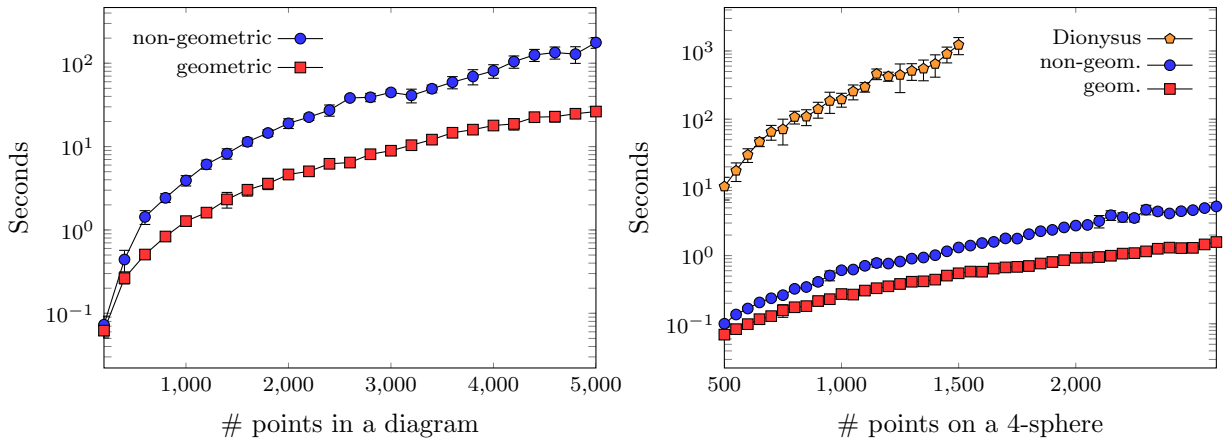


Figure 7: Comparison of non-geometric and geometric variants of the auction algorithm on normal (left) and real (right) input, also with Dionysus on the real input.

an exact version of the auction algorithm. A naive approach to get the exact result is to rescale the input to integer coordinates and choose ϵ such that the approximation error is smaller than 1. We plan to investigate different possibilities to compute the exact distance more efficiently.

5 Conclusion

We have demonstrated that geometry helps to compute bottleneck and Wasserstein distances of bipartite point sets in two dimensions. 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, the auction algorithm offers several variants that can be of interest in practical applications. First, a weighted version of the algorithm allows to assign integer weights to the input points and to match integer fractions of the same bidder to different objects [6]. While this case can be easily reduced to the unweighted case, we expect the weighted version to perform significantly faster. This would lead to an efficient approximation scheme for very large persistence diagrams by simply binning the input points into clusters with multiplicity and computing the Wasserstein distance between the cluster centers. Our preliminary implementation demonstrates that geometry also helps in this weighted setup; we will discuss the details in a full version of this work. We also plan to investigate a parallel version of our auction algorithm.

Acknowledgements

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

Michael Kerber and Arnur Nigmetov acknowledge support by the Max Planck Center for Visual Computing and Communication. Dmitriy Morozov is supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-05CH11231.

References

[1] Aaron Adcock, Daniel Rubin, and Gunnar Carlsson. Classification of hepatic lesions using the matching metric. *Computer Vision and Image Understanding*, 121:36–42, 2014.

[2] 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*, pages 555–564, 2014.

[3] Alexander Andrievsky and Andrei Sobolevskii. WANN: An implementation of weighted nearest neighbor search. Manual, available at `http://www.mccme.ru/~ansobol/otarie/software.html`.

[4] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[5] Dimitri Bertsekas. A distributed algorithm for the assignment problem. Technical report, Laboratory for Information and Decision Sciences, MIT, 1979.

[6] Dimitri Bertsekas and David Castañon. The auction algorithm for the transportation problem. *Annals of Operations Research*, 20(1):67–96, 1989.

[7] Dimitri Bertsekas and David Castañon. Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Computing*, 17(6):707–732, 1991.

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

[9] 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*, pages 318–325, 1992.

[10] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. Stability of persistence diagrams. *Discrete & Computational Geometry*, 37(1):103–120, 2007.

[11] David Cohen-Steiner, Herbert Edelsbrunner, John Harer, and Yuriy Mileyko. Lipschitz functions have L_p-stable persistence. *Foundations of Computational Mathematics*, 10(2):127–139, 2010.

[12] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2nd edition, 2000.

[13] Herbert Edelsbrunner and John Harer. *Computational Topology. An Introduction.* Amer. Math. Soc., 2010.

[14] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 454–463, 2000.

[15] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[16] Alon Efrat, Alon Itai, and Matthew J. Katz. Geometry helps in bottleneck matching and related problems. *Algorithmica*, 31(1):1–28, 2001.

[17] Jennifer Gamble and Giseon Heo. Exploring uses of persistent homology for statistical analysis of landmark-based shape data. *Journal of Multivariate Analysis*, 101(9):2184 – 2199, 2010.

[18] Chen Gu, Leonidas J. Guibas, and Michael Kerber. Topology-driven trajectory synthesis with an example on retinal cell motions. In *International Workshop on Algorithms in Bioinformatics (WABI)*, pages 326–339, 2014.

[19] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[20] Dmitriy Morozov. Dionysus library for computing persistent homology. `mrzv.org/software/dionysus`.

[21] David M. Mount and Sunil Arya. ANN: A library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN`.

[22] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, March 1957.

[23] Pravin M. Vaidya. Geometry helps in matching. *SIAM J. Comput.*, 18(6):1201–1225, 1989.