MOVE SCHEDULES: FAST PERSISTENCE COMPUTATIONS IN SPARSE DYNAMIC SETTINGS *

MATT PIEKENBROCK† AND JOSE A. PEREA‡

Abstract. The standard procedure for computing the persistent homology of a filtered simplicial complex is the matrix reduction algorithm. Its output is a particular decomposition of the total boundary matrix, from which the persistence diagrams and generating cycles can be derived. Persistence diagrams are known to vary continuously with respect to their input; this motivates the algorithmic study of persistence computations for time-varying filtered complexes. Computationally, simulating persistence dynamically can be reduced to maintaining a valid decomposition under adjacent transpositions in the filtration order. In practice, the quadratic scaling in the number of transpositions often makes this maintenance procedure slower than simply computing the decomposition from scratch, effectively limiting the application of dynamic persistence to relatively small data sets. In this work, we propose a coarser strategy for maintaining the decomposition over a discrete 1-parameter family of filtrations. Our first result is an analysis of a simple linear-time strategy which reduces the number of column operations needed to simulate persistence across a fixed homotopy by at most a factor of 2. We then show a modification of this technique which maintains only a sublinear number of valid states, as opposed to a quadratic number of states, and we provide tight lower bounds for this technique. Finally, we provide empirical results suggesting that the decrease in operations needed to compute diagrams across a family of filtrations is proportional to the difference between the expected quadratic number of states, and the proposed sublinear coarsening.

Key words. example, LaTeX

AMS subject classifications. 68Q25, 68R10, 68U05

1. Introduction.

1.1. Overview. The starting point to the work described in this paper is the stability results established in [20, 21]. Indeed, given a triangulable topological space equipped with a sufficiently tame continuous function, persistent homology captures the changes in topology across the sublevel sets of the space, and encodes them in a persistence diagram. The stability of persistence contends that small changes in the underlying function produce small changes in the corresponding diagrams: if the function changes continuously, so too will the points on the persistence diagram. This motivates the study and application of persistence to time-varying or dynamic settings, like that of dynamic metric spaces [30]. As persistence-related computations tend to exhibit high algorithmic complexity (e.g. essentially cubic[1] in the size of the underlying filtration [38]), their adoption to dynamic settings poses a challenging computational problem. With the current state of the art tools, there is no recourse to the problem of computing persistence from a time-varying filtration containing millions of simplices at thousands of snapshots in time. An efficient means of updating the persistence of a [time-varying] filtration has far-reaching consequences: methods that vectorize persistence diagrams for machine learning purposes, like adaptive template functions [43] and persistence images [1], or summary statistics like α-smoothed Betti curves [45], all immediately become computationally viable tools in dynamic settings.

*Submitted to the editors DATE.

Funding: This work was partially supported by the National Science Foundation through grants CCF-2006661 and CAREER award DMS-1943758.

†Department of Computational Mathematics, Science & Engineering, Michigan State University.

‡Department of Mathematics, Michigan State University.

[1]For finite fields, it is known that the persistence computation reduces to the PLU factorization problem, which takes $O(m^\omega)$ where $\omega \approx 2.373$ is the matrix multiplication constant.

Cohen-Steiner et al. refer to a continuous 1-parameter family of persistence diagrams as a *vineyard*, and they give in [21] an efficient algorithm for their computation given a time-varying filtration. The vineyards approach can be interpreted as an extension of the *matrix reduction* algorithm [47], which computes the persistence diagram of a fixed filtration K with m simplices in $O(m^3)$ time. The main step in the reduction algorithm is a specific decomposition $R = DV$ of the boundary matrix D of K. Since V is always full rank (and upper-triangular), an alternative is to compute a $D = RU$ decomposition, where $U = V^{-1}$. The vineyards algorithm transforms a time-varying filtration into an ordered set of adjacent transpositions to be applied to the rows and columns of the decomposition $D = RU$, each of which take at most $O(m)$ time to execute. If one is interested in understanding how the persistent homology of a continuous function changes over time, then this algorithm is sufficient, for homological critical points can only occur when the filtration order changes. Indeed, this algorithm is also efficient asymptotically: if there are d time-points where this filtration order changes, then the vineyards algorithm takes $O(m^3 + md)$ time: one $O(m^3)$-time reduction upfront for the filtration at time t_0 followed by one $O(m)$ operation to update the decomposition for the remaining time points $(t_1, t_2, \ldots, t_d)$. When $d >> m$, this $O(md)$ approach is far more efficient than the $O(dm^3)$ strategy of computing the diagrams independently at every time point, what we call the "naive" approach.

Despite its efficient in theory, vineyards is often not the method of choice in practice in dynamic settings. While there is an increasingly rich ecosystem of software packages offering variations of the standard reduction algorithm (e.g. Ripser, PHAT, Dionysus, etc... see [40] for an overview), implementations of the vineyards algorithm are relatively uncommon.[2] The reason for this disparity is perhaps explained by Lesnick and Wright [35]: "While an update to an RU decomposition involving few transpositions is very fast in practice... many transpositions can be quite slow... it is sometimes much faster to simply recompute the RU-decomposition from scratch using the standard persistence algorithm." Indeed, whether performing the reductions naively or updating the decomposition dynamically using vineyards, they observed that obtaining the decomposition is the most computationally demanding aspect of their 2-D persistence algorithm.

This work seeks to further understand and remedy this discrepancy: building on the work presented in [13], we introduce a coarser approach to the vineyards algorithm. While the vineyards algorithm is efficient at constructing a *continuous* time family of diagrams, it is not necessarily efficient when the time parameter is coarsely discretized. Our methodology is based on the observation that practitioners often don't need (or want!) all d persistence diagrams generated by a continuous 1-parameter of filtrations; usually just $n << d$ of them are sufficient. By exploiting the "donor" concept introduced in in [13], we are able to make a tradeoff in the number of times the decomposition is restored to a valid state, dramatically reducing the total number of column operations needed to apply an arbitrary permutation to the filtration. This tradeoff, paired with a fast greedy heuristic explained in section 3.3.3, yields an algorithm that can update a $R = DV$ decomposition more efficiently than vineyards in sparse contexts, making dynamic persistence more computationally tractable for a wider class of use-cases.

[2] Dionysus 1 has an implementation of vineyards, but the algorithm was never ported to the 2nd version.

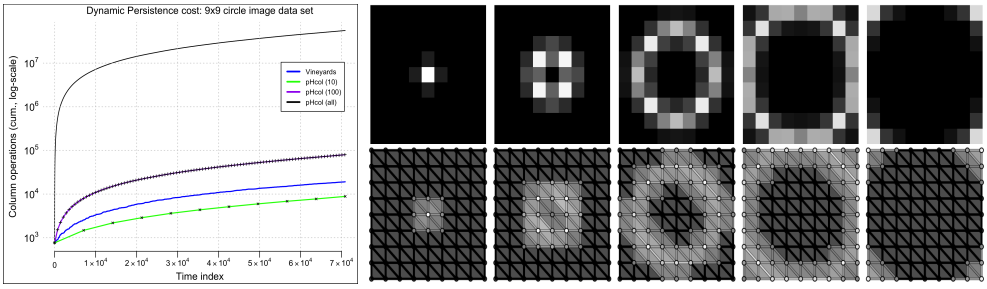


Figure 1.1: On the right, the grayscale image data set and corresponding lower star filtrations. On the left, the cumulative number of column operations required to compute persistence for this time-varying filtration. The persistent Betti numbers for each filtration from left to right are: $\beta_1 = (1,0,0), \beta_2 = (1,1,0), \beta_3 = (1,1,0), \beta_4 = (1,1,0), \beta_5 = (4,0,0)$. Observe computing 10 diagrams independently at evenly spaced time points (the green line) both captures these major topological changes and is the most computationally efficient approach shown.

1.2. A Motivating Example and Contributions. As a motivating example, we describe a simple experiment illustrating why the vineyards algorithm does not always yield an efficient method in many time-varying applications. Consider a series of grayscale images (i.e. a video) depicting a fixed-width annulus expanding about the center of a 9×9 grid, where scale of the annulus is parameterized by a single parameter τ. Thus, the data set is a 1-parameter family of images. Each image contains the same number of pixels whose intensities—these are determined by the parameter—vary with time. For each image, we build a filtered simplicial complex using the *Freudenthal* triangulation, with the ordering of inclusions obtained by lower stars. Snapshots of the image data and its corresponding time-varying filtration is depicted in Figure 1.1 (top right and bottom right, respectively), which we use as a baseline for comparing the standard algorithm `pHcol` with vineyards. Note the size of the filtration is fixed–no simplices are ever added or removed over time. There are two events which dramatically alter the persistence diagrams extracted from these filtrations: the first occurs when the central connected component splits to form a cycle, and the second occurs when the annulus approaches the outer dimensions of the grid, splitting into four components. As a consequence, in this example, only a few persistence diagrams are needed to capture the major topological changes over time.

Consider the situation where one is concerned with capturing these dramatic changes in homology. On the left side of Figure 1.1, we compare the cumulative cost (in total number of column operations) of various approaches which one can use to simulate persistence dynamically. Since it is unknown a priori at what points in time the persistent homology will change in application settings, one solution is to discretize the time domain at n evenly spaced points and compute the persistence diagram at each of these time points independently. An alternative approach is to construct a homotopy between two time-points, and then decompose this homotopy into adjacent transpositions–the vineyards approach. The former is often used in practice, but it is only an approximation; the latter is guaranteed to capture all homological changes in persistence that occur in simulating the 1-parameter family. The graph in Figure 1.1

depicts two different size approximations (purple, $n = 100$ and green, $n = 10$), wherein the reduction algorithm (`pHcol`) was applied at n time points, and two exact strategies (black and blue), which compute all $\approx 7\times10^4$ diagrams given by the homotopy. As one can see from the figure, if one needs to simulate the homotopy exactly, the vineyards approach is indeed far more efficient than naively applying the reduction algorithm independently at all time points. However, when the discretization of the time domain is sparse enough, the naive approach actually performs less column operations than the vineyards strategy, while still capturing the main events.

The existence of a time discretization that is more efficient to compute than continually updating the decomposition indicates that the vineyards algorithm must incur some amount of overhead (in terms of column operations) to perform the transpositions. That is, even in situations where one is updating a decomposition to a filtration that is relatively "close", e.g. any two adjacent filtrations in the bottom right of Figure 1.1, it is still more efficient to apply `pHcol` directly than to iteratively update the decomposition (see the case where $n = 10$). Indeed, any optimizations to the reduction algorithm (for example, the clearing optimization [17]) would only increase this disparity, making the vineyards algorithm less computationally attractive even in settings where n is larger. Since dynamic or time-varying are common settings where persistence is applied (see [44, 46, 35, 30]), it is worth studying this phenomenon.

Our results and approach are as follows: First, we leverage work of Busaryev et. al. [13] to include a coarser type of operation to the dynamic setting, called *moves*. We give a tight lower bound on the number of moves needed to perform an arbitrary permutation to the $R = DV$ decomposition, and give a proof of optimality by a reduction to the permutation edit distance problem. We also give worst-case bounds in expectation as well as efficient algorithms for achieving these bounds—both of which are derived from a reduction to the Longest Increasing Subsequence (LIS) problem. This reduction yields an efficient algorithm for generating sequences of moves $(s_1, s_2, \ldots, s_d)$, which we call *schedules*. We investigate the feasibility of optimizing the cost (i.e., number of column operations) of these schedules directly, including the existence of a greedy-type approach, which we show can lead to arbitrarily bad behavior. In light of these results, we give an alternative proxy-objective to minimize, provide bounds justifying its relevance to the original objective, and give an efficient $O(d \log d)$ algorithm for approximately solving this proxy minimization. A performance comparison with other reduction-based persistent homology computations is given, wherein our approach is demonstrated to be an order of magnitude more efficient than existing approaches in applications such as flock simulation and 2D persistence computation. We conclude with a discussion of situations where our strategy is most applicable and future directions for research.

1.3. Related Work. To the authors knowledge, work focused on ways of updating a decomposition $R = DV$, for all homological dimensions, is limited—there is the vineyards algorithm [21] and the moves algorithm [13], both of which are discussed extensively in section 2.

In contrast, there is extensive work on improving the efficiency of computing a (static) $R = DV$ decomposition. Chen [17] proposed *persistence with a twist*, also called the *clearing optimization*, which exploits a boundary/cycle relationship to "kill" columns early in the reduction rather than reducing them. Another popular optimization is to utilize the duality between homology and cohomology [23], which dramatically improves the effectiveness of the clearing optimization [5]. There are

many other optimizations on the implementation side: the use of ranking functions defined on the combinatorial number system enables implicit cofacet enumeration, removing the need to store the boundary matrix explciitly; the apparent/emergent pairs optimization identifies columns whose pivot entries are unaffected by the reduction algorithm, reducing the total number of columns which need be reduced; sparse data structures such as bit-trees and lazy heaps allow for efficient column-wise additions with $\mathbb{Z}_2 = \mathbb{Z}/2\mathbb{Z}$ coefficients and effective $O(1)$ pivot entry retrieval, and so on [5, 6].

By making stronger assumptions on the underlying topological space, restricting the homological dimension, or targeting a weaker invariant (e.g. Betti numbers), one can usually obtain faster or alternative approaches recording information related to persistence. For example, Attali et al. [2] give a linear time algorithm (under the `word RAM` model of computation) for computing persistence on graphs. In the same paper, they describe how to obtain ϵ-simplifications of 1-dimensional persistence diagrams for filtered 2-manifolds by using duality and symmetry theorems. Along a similar vein, Edelsbrunner et. al. [26] give a fast incremental algorithm for computing persistent Betti numbers up to dimension 2, again by utilizing symmetry, duality, and "time-reversal" [24]. Chen et. al. [18] give an output-sensitive method for computing persistent homology, utilizing the property that certain submatrices of D have the same rank as R, which they exploit through fast sub-quadratic sparse-matrix rank algorithms.

If zeroth homology is the only dimension of interest, computing and updating both the persistence and rank information of the persistent homology groups is greatly simplified. For example, if the relations are available a-priori, obtaining a tree representation fully characterizing the connectivity of the underlying space (also known as the *incremental connectivity* problem) takes just $O(\alpha(n)n)$ time using the disjoint-set data structure, where $\alpha(n)$ is the extremely slow-growing inverse Ackermann function. Adapting this approach to the time-varying setting, Oesterling et al. [39] give a $O(e)$-per-update algorithm that allows one to maintain a *merge tree* with e edges, associated to the filtration parameter changing continuously over time. If only the zeroth-dimensional Betti numbers are needed for a particular application, this problem reduces even further to tracking the connected components of a dynamic graph—sometimes referred to as the *dynamic connectivity problem*, which can be efficiently solved in amortized $O(\log n)$ query and update times using either Link-cut trees or multi-level Euler tour trees [29].

1.4. Outline. The remainder of the paper is organized as follows: we review and establish the notation we will use to describe simplicial complexes, persistent homology, and dynamic persistence. We also cover the reduction algorithm (designated here as `pHcol`), the vineyards algorithm, and the set of *move*-related algorithms introduced in [13], which serves as the starting point of this work. In section 3 we introduce our proposed alternative to vineyards and provide efficient algorithms to make this alternative viable. In section 4 we demonstrate a few applications of the proposed method. In section 5 we conclude the paper by discussing other possible applications and future work. We also include an appendix which introduces certain topics in more detail, such as the vineyards algorithm itself, for readers who may be unfamiliar but would like a more in-depth understanding.

2. Background.

2.1. Notation. Suppose one has a family $\{K_i\}_{i\in I}$ of simplicial complexes indexed by a totally ordered index set I, and so that for any $i < j \in I$ we have

$K_i \subseteq K_j$. There are two index sets of interest in this effort: $\mathbb{R}$ and $[n] = \{1, \ldots, n\}$. Such a family is called a *filtration*, which is deemed to be *essential* if $i \neq j$ implies $K_i \neq K_j$. Moreover, an essential filtration is said to be *simplexwise* if $K_j \setminus K_i = \{\sigma_j\}$ for all $i < j \in I$ with $K_j \neq \emptyset$. Any filtration may be trivially converted into an essential, simplexwise filtration via a set of *condensing*, *refining*, and *reindexing* maps (see [5] for more details). As a result, and without loss of generality, here we exclusively consider essential simplexwise filtrations. For brevity, we will simply refer to them as filtrations.

Let K be an abstract simplicial complex. With respect to some field $\mathbb{F}$, a p-chain is a formal $\mathbb{F}$-linear combination of p-simplices of K. The collection of p-chains under addition yields an abelian group called the p-th chain group of K, denoted $C_p(K)$. The p-boundary $\partial_p(\sigma)$ of a p-simplex σ is the alternating sum of its oriented co-dimension 1 faces; the p-boundary of a p-chain is the sum of the boundaries of its simplices. A p-cycle is a p-chain with zero boundary. The collection of p-cycles forms the group $Z_p(K) = \operatorname{Ker} \partial_p$, the collection of p-boundaries forms the group $B_p(K) = \operatorname{Im} \partial_{p+1}$, and the quotient of these two yields the p-th homology group, $H_p(K) = Z_p(K)/B_p(K)$. If $\{K_i\}_{i \in [m]}$ is a filtration, then the inclusion maps $K_i \subset K_j$ for $i < j$, induce homomorphisms $f_p^{i,j} : H_p(K_i) \to H_p(K_j)$ between their corresponding homology groups:

$$H_p(K_1) \to H_p(K_2) \to \cdots \to H_p(K_m) \tag{2.1}$$

The p-th persistent homology groups are the images of these homomorphisms: $H_p^{i,j} = \operatorname{Im} f_p^{i,j}$. Note that if $i = j$, then $H_p^{i,j} = H_p(K_i) = H_p(K_i)$ is just the "standard" homology. It is common for simplices which create new homology classes to be called *creators* and simplices that destroy homology classes to be called *destroyers*. Similarly, the filtration indices of these creators/destroyers are referred to as *birth* and *death* times, respectively. If a homology class $[c]$ is born at time i and dies entering time j, the difference $|i - j|$ is called the *persistence* of that class. In practice, filtrations often arise from triangulations parameterized by geometric scaling parameters, and the "persistence" of a homology class actually refers to its lifetime with respect to the scaling parameter.

Example 2.1: Non-essential filtrations often arise in geometrical contexts. For example, given a finite metric space (X, d), the *Vietoris-Rips* complex at scale $\epsilon \in \mathbb{R}$ is the abstract simplicial complex given by:

$$\operatorname{Rips}_\epsilon(X) = \{S \subseteq X : S \neq \emptyset \text{ and } \operatorname{diam}(S) \leq \epsilon\} \tag{2.2}$$

The corresponding filtration is called the *Vietoris-Rips* (VR) filtration, indexed by $I = \mathbb{R}$. A VR filtration indexed over $\mathbb{R}$ can be condensed to an essential filtration indexed over the set of pairwise distances $\{\, d(x, x') \mid x, x' \in X \,\}$, which can then be further reindexed to an essential simplexwise filtration by extending a total order of X to its power set using the lexicographical ordering.

Given a triangulable topological space $\mathbb{X}$ equipped with a real-valued function $f : \mathbb{X} \to \mathbb{R}$, we write $\mathbb{X}_a = f^{-1}(-\infty, a]$ to denote the sublevel sets of $\mathbb{X}$ defined by the value a. A *homological critical value* of f is any value $a \in \mathbb{R}$ such that the homology of the sublevel sets of f changes at a, i.e. the inclusion-induced map $H(\mathbb{X}_{a-\epsilon}) \to H(\mathbb{X}_a)$ is not an isomorphism. If there are only finitely many of these homological critical values, f is said to be *tame*.

Algorithm 2.1 Reduction Algorithm (`pHcol`)

Require: $D = (m \times m)$ filtration boundary matrix
Ensure: R is reduced, V is full rank upper triangular, and $R = DV$

```
function REDUCTION(D)
    (R, V) ← (D, I)
    for j = 1 to m do
        while ∃ i < j such that low_R(i) = low_R(j) do
            λ ← pivot_R(j)/pivot_R(i)
            (col_R(j), col_V(j)) −= (λ · col_R(i), λ · col_V(i))
    return (R, V)
```

Consider a homotopy $F(x, \tau) : \mathbb{X} \times [0, 1] \to \mathbb{R}$ and denote it's "snapshot" at a given time-point τ by $f_\tau(x) = F(x, \tau)$. The snapshot f_0 denotes the initial function at time $\tau = 0$ and f_1 denotes the function at the last time step. As τ varies in $[0, 1]$, the points in $\mathrm{dgm}_p(f_\tau)$ trace curves in $\bar{\mathbb{R}}^3$, called *vines* and which together form a *vineyard*. The stability of persistence implies that these curves will be continuous if the homotopy is continuous. The vineyard analogy acts as a guidepost for practitioners seeking to understand how subtle changes occurring to the topological structure over time reveal intrinsic information about the underlying continuous process. We discuss the vineyards approach more in detail in section 2.3.

2.2. The Reduction Algorithm. In this section we briefly recount the original reduction algorithm introduced in [47], also sometimes called the *standard* algorithm or more explicitly `pHcol` [23]. The pseudocode is outlined in Algorithm 2.1. Without optimizations, like the clearing optimization or the use of implicit matrix reduction, the standard algorithm is very inefficient. Nonetheless, the reduction algorithm serves as the foundation of most implementations that compute persistent homology, and its invariants are necessary before introducing both vineyards in section 2.3 and our approach in section 3.

The main output of the reduction algorithm is a matrix decomposition $R = DV$, where the persistence diagram is encoded in R. Given a filtration K with m simplices and maximal dimension d, one assembles the elementary boundary chains $\partial(\sigma)$ as columns ordered according to the filtration order, forming the *filtration boundary matrix* D. The reduction algorithm can compute the persistent homology for all dimensions up to $d - 1$. In this case, D is a square matrix of size $m \times m$. Alternatively, a single dimension $i \leq d - 1$ can be computed, in which case one reduces a pair of matrices D_i, D_{i+1}, where the former has dimension $d_{i-1} \times d_i$ and the latter $d_i \times d_{i+1}$. If one is interested in a particular dimension of homology then the latter approach is preferred. However, the algorithmic description of the algorithms is simpler when considering the full matrix, so D will always be $(m \times m)$ in the descriptions that follow. All algorithms discussed generalize to both situations. We give an example using the matrix pair (R_1, V_1) below.

Example 2.1: Reduction Consider a triangle with vertices u, v, w and edges $a = (u, w)$, $b = (v, w)$, $c = (u, v)$ whose filtration order, read as inclusions from left to right, is given as $K = (u, v, w, a, b, c)$. The reduction algorithm begins by setting $R = D$ and $V = I$, followed by left-to-right column operations until invariant (2)

below is satisfied. If the i-th column of R is nonzero, then $\text{low}_R(i)$ is the row index of its lowest nonzero entry. Using $\mathbb{Z}_2$ coefficients and homology in dimension 1 to simplify the presentation, the steps look at follows:

$$\begin{array}{c} R_1 \; a \; b \; c \\ u \\ v \\ w \end{array}\!\!\begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ 1 & 1 & \end{bmatrix}\!\!\begin{array}{c} V_1 \; a \; b \; c \\ a \\ b \\ c \end{array}\!\!\begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \rightarrow \begin{array}{c} R_1 \; a \; b \; c \\ u \\ v \\ w \end{array}\!\!\begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ 1 & & \end{bmatrix}\!\!\begin{array}{c} V_1 \; a \; b \; c \\ a \\ b \\ c \end{array}\!\!\begin{bmatrix} 1 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} \rightarrow \begin{array}{c} R_1 \; b \; a \; c \\ u \\ v \\ w \end{array}\!\!\begin{bmatrix} & 1 & \\ 1 & 1 & \\ 1 & & \end{bmatrix}\!\!\begin{array}{c} V_1 \; b \; a \; c \\ b \\ a \\ c \end{array}\!\!\begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix}$$

The column c in R indicates that dimension 1 homology is born, which in this filtration is never killed since the dimension of the complex is 2. Similarly, the columns at u, v, w in R_0 (not shown) are all zero, indicating three 0-dimensional homology classes are born, which are then killed by the pivot entries in columns a and b in R_1.

Its clear from Algorithm 2.1 that a loose upper bound for this reduction is $O(m^3)$, where m is the number of simplices of the filtration; it turns out that this bound is in fact tight, see [38] for more details. There are actually many variations and optimizations proposed over the past decade to Algorithm 2.1; for example, one can also use the *row algorithm* [23], or reverse the order of the filtration and compute cohomology [23], etc. Despite these variations in computation, regardless of how the decomposition is obtained, it must obey the following invariants:

Decomposition Invariants:

1. $R = DV$ where D is the boundary matrix of the filtration K
2. V is full-rank upper-triangular, and R is *reduced*: $\text{low}_R(i) \neq \text{low}_R(j)$ if its i-th and j-th columns are nonzero

If these two invariants are maintained, then we call the decomposition *valid*. Given any $R = DV$ decomposition, it was shown in [47] that the following proposition is true: Amongst other important corollaries, one immediate consequence of this proposition is that the persistent pairing is independent of the particular entries in R and V. When reducing the boundary matrix, we can therefore perform column operations in any order, as long as columns are added from left to right, and so long as every column operation performed in R is also performed on V. Once R is reduced, if both of the invariants are respected, then the low entries in R yield the correct pairing encoding the persistence diagram of the corresponding filtration K.

2.3. Vineyards. The original purpose of the vineyards algorithm, as described in [21], was to compute a continuous 1-parameter family of persistence diagrams given a time-varying filtration, and to detect homological critical events during the construction. Notice homological critical events can only occur when the filtration order changes, though not all changes in the filtration order result in homology changes. It follows that detecting these homological events reduces to computing matrix decompositions at a finite set of time points interleaved between homology-altering changes. At the finest scale, changes to the filtration order manifest as adjacent transpositions. As a result, a fixed set of rules for maintaining a valid $R = DV$ decomposition under the transposition of adjacent simplices in the filtration is enough to simulate persistence dynamically. These rules prescribe certain column and row operations which must be applied to the matrix decomposition either before, during, or after the transposition, in order to ensure decomposition invariants (1) and (2) are respected.

Let S_i^j represent the upper-triangular matrix such that multiplying by it on the right is equivalent to adding column i to column j. Multiplication by S_i^j on the left is equivalent to adding row j to row i, and thus $S_i^j S_i^j = I$. Similarly, let P denote the permutation matrix so that multiplication from the right applies some permutation p to the columns of some matrix A. The inverse of any permutation is also its transpose, i.e. $P^{-1} = P^T$, and thus one would write PAP^T to denote the application of the permutation P to both the columns and rows of A. In the special case where P represents a transposition, we have $P = P^T$ and may instead simply write PAP. The goal of the vineyards algorithm can now be described explicitly: to prescribe a set of rules, written as matrices S_i^j, such that if $R = DV$ is a valid decomposition, then $(*P * R * P*) = (PDP)(*P * V * P*)$ is also a valid decomposition, where $*$ is some number (possibly zero) of matrices encoding elementary column or row operations. The explicit pseudocode describing these rules is given in the appendix by Algorithm .1.

Example 2.2 To illustrate the basic principles on which vineyards works, we reuse the running example introduced in the previous section. Below, we illustrate the case of exchanging simplices a and b in the filtration order.

$$\begin{array}{c} R_1 \begin{matrix} a & b & c \end{matrix} \\ \begin{matrix} u \\ v \\ w \end{matrix} \begin{bmatrix} 1 & 1 & \\ & 1 & \\ 1 & & \end{bmatrix} \end{array}, \begin{array}{c} V_1 \begin{matrix} a & b & c \end{matrix} \\ \begin{matrix} a \\ b \\ c \end{matrix} \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array} \xrightarrow{S_1^2} \begin{array}{c} \begin{matrix} a & b & c \end{matrix} \\ \begin{bmatrix} 1 & & \\ & 1 & \\ 1 & 1 & \end{bmatrix} \end{array}, \begin{array}{c} \begin{matrix} a & b & c \end{matrix} \\ \begin{bmatrix} 1 & 1 & \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array} \xrightarrow{P} \begin{array}{c} \begin{matrix} b & a & c \end{matrix} \\ \begin{bmatrix} 1 & & \\ & 1 & \\ 1 & 1 & \end{bmatrix} \end{array}, \begin{array}{c} \begin{matrix} b & a & c \end{matrix} \\ \begin{bmatrix} 1 & 1 & \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array} \xrightarrow{S_1^2} \begin{array}{c} R_1 \begin{matrix} b & a & c \end{matrix} \\ \begin{matrix} u \\ v \\ w \end{matrix} \begin{bmatrix} 1 & & \\ 1 & 1 & \\ 1 & & \end{bmatrix} \end{array}, \begin{array}{c} V_1 \begin{matrix} b & a & c \end{matrix} \\ \begin{matrix} b \\ a \\ c \end{matrix} \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}$$

Prior to performing the exchange, observe that that the highlighted entry in V_1 would render V_1 non-upper triangular after the exchange. This entry is removed by a left-to-right column operation, given by applying the S_1^2 on the right to R_1 and V_1. After this operation, the permutation may be safely applied to V_1. In this case, both before and after the permutation, R_1 is rendered non-reduced, requiring another column operation to restore the decomposition to a valid state.

The time complexity of vineyards is determined entirely by the complexity of performing a single adjacent transposition, and its complexity is often stated in this way. Informally, since column operations are the largest complexity operations needed and each column can have potentially m entries, the complexity of vineyards is $O(m)$ per transposition. However, there are a number of subtleties involved in this statement. The first subtlety is that both the V and R matrices tend to be quite sparse, and thus column and row operations rarely require m operations—in fact, as a rule of thumb, most transpositions require no column operations [26]. A second subtlety, due to the necessity of sparse matrix data structures, is that querying the non-zero status of any particular entry typically takes $O(\log m)$ time. This hidden cost may seem minor, however, the quadratic scaling of the number of transpositions vineyards typically requires makes this cost non-trivial. In fact, achieving $O(m)$ complexity per transposition requires a special sparse matrix representation that allows swapping any two rows and columns in $O(1)$—see the appendix for more details and discussion. Finally, note the constant factor associated with the stated complexity: inspection of the individual cases of Algorithm .1 shows that there are at most two $O(m)$ operations, on both R and V, that may be needed for any single transposition.

2.4. Moves. Busaryev et al. [13] introduced an algorithm which maintains an $R = DV$ decomposition under *move operations*. A move operation $\text{Move}(i, j)$ is a set

of rules for maintaining a valid decomposition under the application of a permutation P that moves a simplex σ_i at position i to position j. If $j = i \pm 1$, this operation obviously is just an adjacent transposition, and thus in some sense Busaryev's move algorithm is a generalization of the vineyards algorithm.

Though they are similar, move operations confer a few computational advantages compared to vineyards. Informally, a move operation has two convenient properties:

1 Querying the non-zero status of entries in any matrix representation occurs once per move.

2 The decomposition $R = DV$ is not maintained during the movement of $\sigma_i \mapsto \sigma_j$.

First, consider property (1). Prior to applying any permutation P to the decomposition, it is necessary to remove non-zero entries in V which render PVP^T non-upper triangular, to maintain invariant (2). If one uses the (augmented) sparse matrix representation described in the original vineyards publication [21], checking whether these entries are non-zero takes $O(\log m)$ per column. Since moving a simplex from i to j using vineyards performs $|i-j|-1$ transpositions, this query operation is required at most $O(|i-j|)$ times, which is excessive. If a row-oriented sparse matrix implementation is used, a move operation can perform these non-zero entry checks in just one $O(m)$ pass, prior to performing any column operations.

Property (2) indicates move operations only guarantee the decomposition is valid after the operation completes, implying that the decomposition is not fully maintained during the execution of *Restore*$*$. This is a significant difference compared to the vineyards algorithm, which was designed specifically to maintain a valid decomposition after each transposition. Thus, move operations can be thought of as making a tradeoff in granularity: whereas a sequence of adjacent transpositions $(i, i{+}1), (i{+}1, i{+}2), \ldots, (j{-}1, j)$ generates $|i-j|$ valid decompositions in vineyards, a move operation Move(i, j) generates only one.

If one has a pair of filtrations (K_0, K_1) each with m simplices, and we assume that each simplex σ_i switches its relative ordering with another simplex σ_j at most once in some ordered schedule of transpositions, then the number of intermediate persistence diagrams is bounded above by $O(m^2)$ using transpositions, whereas with move operations this bound clearly is at most $O(m)$.

Example: We continue the example used in sections 2.3 and 2.2 to illustrate moves. Consider moving the edge a to the position taken by the edge c in the filtration. The donor columns introduced in [13] are shown on the left side of each matrix. Note that using vineyards, the equivalent permutation to the decomposition would require 4 column operations on both R_1 and V_1, respectively, whereas a single move operation accomplishes this permutation using only 2 column operations per matrix.

$$
\begin{array}{c c}
d_R & \begin{array}{c} a \end{array} \\
\begin{array}{c} u \\ v \\ w \end{array} & \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix}
\end{array}
\quad
\begin{array}{c c}
R_1 & \begin{array}{c} a\ b\ c \end{array} \\
\begin{array}{c} u \\ v \\ w \end{array} & \begin{bmatrix} 1 & 1 & \\ & & 1 \\ 1 & & \end{bmatrix}
\end{array}
\rightarrow
\begin{array}{c c}
 & \begin{array}{c} b \end{array} \\
 & \begin{bmatrix} 1 \\ 1 \\ \end{bmatrix}
\end{array}
\begin{array}{c c}
 & \begin{array}{c} a\ b\ c \end{array} \\
 & \begin{bmatrix} 1 & & \\ & 1 & \\ 1 & 1 & \end{bmatrix}
\end{array}
\rightarrow
\begin{array}{c c}
 & \begin{array}{c} c \end{array} \\
 & \begin{bmatrix} \\ \\ \end{bmatrix}
\end{array}
\begin{array}{c c}
 & \begin{array}{c} a\ b\ c \end{array} \\
 & \begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ 1 & 1 & \end{bmatrix}
\end{array}
\xrightarrow{P}
\begin{array}{c c}
 & \begin{array}{c} c \end{array} \\
 & \begin{bmatrix} \\ \\ \end{bmatrix}
\end{array}
\begin{array}{c c}
 & \begin{array}{c} b\ c\ a \end{array} \\
 & \begin{bmatrix} & 1 & 1 \\ 1 & 1 & \\ 1 & & 1 \end{bmatrix}
\end{array}
\xrightarrow{d_R}
\begin{array}{c c}
R_1 & \begin{array}{c} b\ c\ a \end{array} \\
\begin{array}{c} u \\ v \\ w \end{array} & \begin{bmatrix} & & 1 \\ 1 & 1 & \\ 1 & & \end{bmatrix}
\end{array}
$$

Algorithm 2.2 Move Right Algorithm

function RestoreRight(R, V, $\mathbb{I} = \{I_1, I_2, \ldots, I_s\}$)
  $(\, d_{low},\, d_R,\, d_V \,) \leftarrow (\, \mathrm{low}_R(I_1),\, \mathrm{col}_R(I_1),\, \mathrm{col}_V(I_1) \,)$
  for k **in** $I_2, \ldots, I_s$ **do**
    $(\, d'_{low},\, d'_R,\, d'_V \,) \leftarrow (\, \mathrm{low}_R(k),\, \mathrm{col}_R(k),\, \mathrm{col}_V(k) \,)$
    $(\, \mathrm{col}_R(k),\, \mathrm{col}_V(k) \,)$ += $(\, d_R,\, d_V \,)$
    if $d'_{low} < d_{low}$ **then**
      $(\, d_{low},\, d_R,\, d_V \,) \leftarrow (\, d'_{low},\, d'_R,\, d'_V \,)$
  return $(\, R,\, V,\, d_R,\, d_V \,)$

function MoveRight(R, V, i, j)
  $\mathbb{I}$ = columns satisfying $V[i, i:j] \neq 0$
  $\mathbb{J}$ = columns satisfying $\mathrm{low}_R \in [i, j]$ and $\mathrm{row}_R(i) \neq 0$
  $(\, R,\, V, d_R, d_V \,) \leftarrow$ RestoreRight(R, V, $\mathbb{I}$)
  $(\, R,\, V \,) \leftarrow$ RestoreRight(R, V, $\mathbb{J}$)
  $(\, R,\, V \,) \leftarrow (\, PRP^T,\, PVP^T \,)$
  $(\, \mathrm{col}_R(j),\, \mathrm{col}_V(j) \,) \leftarrow (\, Pd_R,\, Pd_V \,)$
  return $(\, R,\, V \,)$

$$
\begin{array}{ccccccccccc}
d_V\ a & & V_1\ a\ b\ c & & b & & a\ b\ c & & c & a\ b\ c & \\
\begin{bmatrix} 1 \\ \\ \end{bmatrix} & \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} & \to & \begin{bmatrix} 1 \\ 1 \\ \end{bmatrix} & \begin{bmatrix} 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} & \to & \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} & \xrightarrow{P} & \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} & \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \\
 & & & & & & & & & & \xrightarrow{d_V} \\
 & & & & & & & & & & \begin{matrix} b \\ c \\ a \end{matrix} \begin{bmatrix} 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix}
\end{array}
$$

We recall an important claim given by [13] on the effect that move operations have on the status of simplices in the pairing. Recall from section 2 that simplices which create new homology classes are called *creators* and simplices that destroy homology classes are called *destroyers*. The effect of the movement on intermediate simplices depends on the direction of the movement. If $i < j$ (respectively, $j < i$), all simplices at positions $k \in [i, j] \setminus i$ are shifted down (respectively, up) by 1. The pseudo-code for the *MoveRight* operation is given in Algorithm 2.2, and the pseudo-code for the *MoveLeft* is given in Algorithm 2.3. We note that while the pseudocode for *MoveRight* is included in [13], the one for *MoveLeft* is not, and Algorithm 2.3 is the one we propose and leverage in this paper.

Although conceptually similar, there is an asymmetry between *MoveRight* and *MoveLeft*. Moving a simplex upwards in the filtration requires removing non-zero entries along several columns of a particular row in V so that the corresponding permutation does not render V non-upper triangular. The key insight of the algorithm presented in [13] is that R can actually be maintained in all but one column during this procedure (by employing the *donor* column). In contrast, moving a simplex to an earlier time in the filtration requires removing non-zero entries along several rows of a particular column of V. As before, R stays reduced during this cancellation procedure in all but one column, however the subsequent permutation to R requires reducing a pair of columns which may cascade into a larger chain of column operations to

Algorithm 2.3 Move Left Algorithm

```
function RestoreLeft(R, V, 𝕂 = {k_1, k_2, ..., k_s})
    (l, r) ← indices l, r ∈ 𝕂 satisfying l < r, low_R(l) = low_R(r) maximal
    while low_R(l) ≠ 0 and low_R(r) ≠ 0 do
        (col_R(r), col_V(r)) += (col_R(l), col_V(l))
        𝕂 ← 𝕂 \ l
        (l, r) ← indices l, r ∈ 𝕂 satisfying l < r, low_R(l) = low_R(r) maximal
    return ( R, V )
function MoveLeft(R, V, i, j)
    𝕁 ← {j}
    while V(k, i) ≠ 0 for k = low_V(i) where j ≤ k < i do
        (col_R(i), col_V(i)) += (col_R(k), col_V(k))
        𝕂 ← 𝕂 ∪ k + 1
    (R, V) ← (PRP^T, PVP^T)
    𝕁 = columns satisfying low_R ∈ [i, j] and row_R(i) ≠ 0
    (R, V) ← RestoreLeft(R, V, 𝕂)
    (R, V) ← RestoreLeft(R, V, 𝕁)
    return ( R, V )
```

keep R reduced. Since these operations always occur in a left-to-right fashion, its not immediately clear how to apply a donor column kind of concept. Fortunately, we can still separate out these two stages.

3. Our contribution: Move Schedules.

3.1. Proposed approach. Let us begin by briefly describing the overall pipeline of our proposed approach, which is outlined in Algorithm 3.1. As before, we assume as input a discrete 1-parameter family of filtrations $\mathcal{K} = (K_1, K_2, \ldots, K_n)$, and the goal being to compute a persistence diagram for each filtration.

Without loss of generality, we may assume $|K_i| = m$ for every $i \in [n]$, such that there are bijections $f_i : K_i \to K_{i+1}$. Otherwise, if $|K_i| \neq |K_{i+1}|$, we may append $K_{i+1} \setminus K_i$ to the end of K_i and likewise append $K_i \setminus K_{i+1}$ to the end of K_{i+1} prior to constructing the bijection. At the matrix level, both concatenations can be achieved by appending the boundary chains of the simplices being inserted to R, and then running Algorithm 2.1 only on the appended columns. Each f_i induces a bijection $f_i^* : [m] \to [m]$, or equivalently, a permutation of the index set $[m]$. For each pair of filtrations (K_i, K_{i+1}), we assign labels to the simplices of K_i using the index set $[m]$ and relabel K_{i+1} accordingly using f_i. Denote these permutations by p and q. We compute the longest increasing subsequence (LIS) of q and use this subsequence to recover a longest common subsequence (LCS) $\mathrm{LCS}(p, q)$. We pass p, q, and this LCS to our scheduling algorithm, which returns as output an ordered set of move permutations. We collect all of these permutations into a single schedule $\mathcal{S}$, obtain an $R = DV$ decomposition for the first filtration K_1, and then execute the permutations stored in $\mathcal{S}$ in sequence. During execution, we obtain a valid decomposition for each filtration $K_1, K_2, \ldots, K_n$. It turns out that not all valid schedules are created equal: some incur a much larger number of column operations, even if they have a minimal

number of moves. Minimizing schedule costs will be addressed in Section 3.3.

Algorithm 3.1 Scheduling algorithm

Require: Ordered set of filtrations $\mathcal{K}$, each of size m, with bijections $f_i : K_i \to K_{i+1}$
Ensure: Valid $R = DV$ decompositions are computed for each filtration $K_1, K_2, \ldots, K_n$

```
1: procedure SparseSchedule(K = (K_1, K_2, ..., K_n))
2:     S = ∅
3:     for i = 1 to n − 1 do
4:         (p, q) ← ([m], Im f_i^*)
5:         lis_q ← LIS(q)                                   ▷ O(m log log m)
6:         S ← S ∪ GreedySchedule(p, q, lis_q)              ▷ O(d log d), see section 3.3.3
7:     (R, V) ← Reduction(D = ∂K_1)
8:     for (i, j) in S do
9:         if i < j then
10:            (R, V) ← MoveRight(i, j)
11:        else
12:            (R, V) ← MoveLeft(i, j)
```

Algorithm 3.1 is purely illustrative at this point, and is meant to serve as a guidepost before discussing its other components in depth. Specifically, lines (4-5) are discussed next in section 3.2, and the algorithm for line (6) is given in section 3.3. Note that the loop starting at line (3) in Algorithm 3.1 could always be merged with the loop starting at line (8), i.e. the individual schedules S_i may be computed on the fly. Due to the combinatorial aspect of our approach, there is actually no need to have access to—or explicitly store—the entire family of filtrations and the schedules between them. That is, Algorithm 3.1 can be easily modified to be completely online, keeping at most two filtrations and one decomposition in memory at any given time.

3.2. Minimizing schedule size. Here we describe the reasoning for lines (4-5) in Algorithm 3.1. Recall in section 1.2, an example was given showing the vineyards algorithm efficiency compared to a naïve approach at three varying levels of coarseness. It was hypothesized that the reason the vineyards algorithm was more expensive than the naïve approach is due to the extra overhead of maintaining the decomposition at each transposition. Thus, decreasing the number of times the decomposition is restored to a valid state ought to reduce the total number of column operations needed to update a decomposition. This motivates the following question: in order to apply an *arbitrary* permutation P to a given $R = DV$ decomposition, what is the minimal number of times the decomposition needs to be restored to a valid state?

3.2.1. The Continuous Setting. In the vineyards case, we are given a homotopy $F(x, \tau) = \mathbb{X} \times [0, 1] \to \mathbb{R}$ and a pair of filtrations (K_0, K_1) whose simplices are ordered according to $f_0(x) = F(x, 0)$ and $f_1(x) = F(x, 1)$, respectively. The choice of homotopy F completely determines the number of adjacent transpositions, so to get a meaningful bound we must know something about F. If we assume the curves traced by the homotopy are in general position and we assume each pair of curves cross each other at most once, then the set of adjacent transpositions S_F that F decomposes

into is given by the pairs of simplices which change in their relative ordering:

$$S_F = \{(i,j) \,|\, f_0(\sigma_i) < f_0(\sigma_j) \wedge f_1(\sigma_i) > f_1(\sigma_j) \quad \forall\, i,j \in [m]\} \tag{3.1}$$

$$= (s_1, s_2, \ldots, s_k) \tag{3.2}$$

We assume, without loss of generality, that the underlying complexes K_0 and K_1 have the same cardinality. Since we deal exclusively with essential filtrations, we may think of K_1 as simply a reordering of K_0 (or vice versa). Let $p = [m]$ and let $q = \operatorname{Im} f^*$, where $f^* : [m] \to [m]$ is the one-to-one correspondence induced by the bijection $f : K_0 \to K_1$. Then S_F is given by inversions between p and q:

$$\operatorname{Inv}(p,q) = \{\, (i,j) \,|\, p(i) < p(j) \wedge q(i) > q(j) \,\}$$

The associated distance, measuring $|S_F|$, is the *Kendall-τ* distance $K_\tau(p,q)$:

$$K_\tau(p,q) = |\operatorname{Inv}(p,q)| \tag{3.3}$$

Geometrically, our assumptions on F defines a class of x-monotone curves called *pseudo-segments*, i.e. parameterized curves whose behavior with respect to a certain restricted set of geometric predicates is invariant. This family includes e.g. the straight-line homotopy $f_\tau(\sigma) = (1-\tau)f_0(\sigma) + \tau f_1(\sigma)$, which was studied in the original vineyards paper [21]. Detecting all k intersections of m pseudo-segments is a well-studied problem in computational geometry that can be optimally solved in output-sensitive $O(m \log m + k)$ time by several algorithms [11], where k is the output-sensitive term. Since $k \sim O(m^2)$ in the worst case, achieved when $f_1 = -f_0$, we have the following upper bound on $|S_F|$

$$|S_F| \approx O(m^2) \tag{3.4}$$

This bound is exhibited in the grayscale image data example in section 1.2: each 9x9 patch contains $(81, 208, 128)$ simplices of dimensions $(0, 1, 2)$ yielding $m = 417$ simplices in total which, as the homotopy is simulated, generate $\approx$ 70,000 transpositions. The upper bound size on $|S_F|$ is thus $\binom{417}{2} \approx$ 87,000, which nearly occurs in this case, since the last filtration is nearly the reverse of the filtration at the beginning. This quadratic scaling can induce a number of issues in practical implementations of vineyards, e.g. although many transpositions require 0 column reductions in practice, detecting the existence of non-zero entries at specific positions in the sparse matrix takes $O(\log(m))$ time. Since this check is required after each permutations are applied, it is in some sense unavoidable in the vineyards algorithm.

The ordered set of inversions $(i,j) \in S_F$ can be interpreted as permutations to apply to the decomposition. We call S_F a *schedule* with respect to F. If our goal is to decrease the size of $|S_F|$, one option is to *coarsen* S_F by collapsing contiguous sequences of adjacent transpositions to moves, via the map:

$$(i,i+1)(i+1,i+2)\cdots(j-1,j) \mapsto (i,i+1,\cdots,j-1,j) \quad \text{if } i < j \tag{3.5}$$

We expect a coarsened schedule to be cheaper to execute than the original schedule it was derived from, with the tradeoff being that less diagrams are produced, which we now prove

PROPOSITION 3.1. *For some given decomposition $R = DV$ where each matrix is $(m \times m)$, let C denote the number of $O(m)$ operations required to execute schedule*

$S = (t_1, t_2, \ldots, t_h)$, and similarly let L be the number of $O(m)$ operations required to execute a coarsened schedule $\widetilde{S}$ of S. We have the following relationship between C and L:

$$\frac{C}{2} \leq L \leq C$$

Proof. Consider Algorithms .1 and 2.2, where it's assumed Algorithms .1 is used to execute a given schedule S and Algorithms 2.2 is used to execute its coarsened schedule $\widetilde{S}$. If maps are created providing $O(1)$ access to the low entries needed by lines (5) and (12), and the non-zero entries in each column are sorted according to some fixed order providing $O(\log(m))$ time for lines (5), (11), (17), and (20), then the dominating cost of each transposition in Algorithm .1 are the column operations, each of which take $O(m)$ time. There are at most 2 column operations for any given case. For any contiguous sequence of transpositions $(t_i, t_{i+1}, \ldots, t_j)$, this implies we have at most $2|i - j|$ operations each requiring $O(m)$ time. Now consider a single move operation, Move(i,j), as a replacement for the sequence of transpositions above. Identifying non-zero entries in the matrix occurs once, which takes $O(|i - j| \log(m))$ time. If we maintain a map providing low entries in $O(1)$ time, since permutations take $O(1)$ time and lines (7) and (7) in both Restore* and Move* takes just $O(1)$ via pointer swapping, the dominant cost again are the column operations (line 5). Since a move operation requires at most a single column operation for each index between $[i, j]$, the complexity of Move(i, j) is bounded above by $O(|i - j|m)$, and the claimed inequality follows. ∎

In terms of bounds, we clearly have $|\widetilde{S}_F| \leq |S_F|$, and the associated coarsened schedule $\widetilde{S}_F$ can be computed in $O(m)$ time, so this is a viable approach to improving the efficiency of the vineyards algorithm. However, the coarsening depends entirely on the choice of F and the upper bound from 3.4 remains, as its always possible that there are no contiguous subsequences to collapse. If a straight line homotopy is used, the number of collapsible subsequences may depend on the simplices categorization as critical points; transpositions between pairs of simplices which trigger a changes in homology tend to be more expensive. Cohen-Steiner [21] referred to these as *switches*, see [26] for more details.

3.2.2. The Discrete Setting. Since moves do not maintain a valid decomposition during movement, it is worth asking if we can achieve better bounds than vineyards, by removing the assumption that the goal is to simulate persistence across a supplied homotopy $F(x, \tau) = \mathbb{X} \times [0, 1] \to \mathbb{R}$. In the case of vineyards, the bound is clear: whether assuming a homotopy between a pair of filtrations (K_0, K_1) or not, if $\kappa = K_\tau(K_0, K_1)$, then $\Omega(\kappa)$ adjacent transpositions are required to transform $K_0 \mapsto K_1$. In contrast, if we allow move operations in any order, we have $O(m)$ as a trivial upper bound: simply move each simplex in K_0 into its position in the filtration given by K_1 in the order given by the latter. However, it's not immediately clear whether this bound is tight. To establish this formally, we require a few definitions.

A *permutation* is a bijection of a set onto itself. The symmetric group S_n is the set of all permutations of $[n]$ under function composition $\circ$, applied from right to left. Given two fixed permutations $p, q \in S_n$ and a set $S \subseteq S_n$, a few goals that appear commonly in the literature are the following:

1 Find an ordered sequence of permutations $s_1, s_2, \ldots, s_d$ from S whose composition transforms p into q:

$$s_d \circ \cdots \circ s_2 \circ s_1 \circ p = q$$

2 Find a sequence satisfying (1) of minimal length (d)
3 Find the minimal length d, referred to as the *distance* between p and q with respect to S.

A sequence $s_1, s_2, \ldots, s_d \in S \subseteq S_n$ of operations mapping $p \mapsto q$ is sometimes called a *sorting of* p in the literature. In the context of performing updates to the $R = DV$ decomposition, we call such sequences *schedules* . There are typically many ways to solve (1), and any solution to (2) necessarily solves (1) and (3). When p, q are interpreted as strings, distances defined with respect to a fixed $S \subseteq S_n$ are commonly referred to as *edit distances* [32], which are denoted as $d_S(\cdot, \cdot)$. Note that here we describe distances between permutations (e.g. $p, q \in S_n$) using operations that can themselves be expressed as permutations (e.g. $s \in S \subseteq S_n$), and that the choice of S defines the universe of operations and thus the type of edit distance being measured—otherwise if $S = S_n$, then $d_S(p, q) = 1$ for any $p, q \in S_n$ and the problem is trivial!

Many types of edit distances can be described succinctly. Let $\beta(i, j, k, l)$ with $1 \leq i < j \leq k < l \leq n$ denote the permutation that exchanges the two closed intervals given by $i, j-1$ and $k, l-1$:

$$\begin{pmatrix} 1 & \cdots & i-1 & \boxed{i} & \boxed{i+1} & \cdots & \boxed{j-1} & j & j+1 & \cdots & k-1 & \boxed{k} & \boxed{k+1} & \cdots & \boxed{l-1} & l & l+1 & \cdots & n \\ 1 & \cdots & i-1 & \boxed{k} & \boxed{k+1} & \cdots & \boxed{l-1} & j & j+1 & \cdots & k-1 & \boxed{i} & \boxed{i+1} & \cdots & \boxed{j-1} & l & l+1 & \cdots & n \end{pmatrix}$$

Common edit distances found in the literature expressed as specializations of β are given below:

1 (no restriction on i, j, k, l) $\implies$ β is called a *block interchange*
2 $j = k$ $\implies$ β swaps adjacent intervals, called a *transposition*
3 $j = i + 1$ and $l = k + 1$ $\implies$ β swaps two not necessarily adjacent elements, called an *exchange*[3]

Restricting the operations given above further to the case where $i = 1$ gives the so-called *prefix* versions of the described distances. Although each edit distance can be described using common notation, both the difficulty of obtaining a minimal sorting and the size $d_S(p, q)$ varies dramatically in the choice of S. For example, while sorting by transpositions and reversals is NP-hard and sorting by prefix transpositions is unknown, there are polynomial time algorithms for sorting by block interchanges, exchanges, and prefix exchanges [32]. Sorting by adjacent transpositions can be achieved in many ways: any sorting algorithm that exchanges two adjacent elements during its execution (e.g. bubble sort, insertion sort, merge sort) yields a sorting of size $K_\tau(p, q)$.

Here we consider sorting by move permutations. A *move* on the set $[m]$ is a permutation given by a pair (i, j), and its effect is to cyclically rotate the interval $[i, j]$. For example, if $i < j$, this permutation takes the form:

$$\begin{pmatrix} 1 & \cdots & i-1 & \boxed{i} & \boxed{i+1} & \cdots & \boxed{j-1} & \boxed{j} & j+1 & \cdots & m \\ 1 & \cdots & i-1 & \boxed{i+1} & \cdots & j-1 & j & i & j+1 & \cdots & m \end{pmatrix}$$

The complexity of sorting by moves is related to the well-known *Levenshtein distance* [4], also known as the edit distance on strings. The Levenshtein distance between two strings A and B is the minimum number of character insertions, deletions, and substitutions needed to transform one string into another. Through a dynamic programming type approach, the Wagner-Fischer algorithm can compute the Levenstein distance in $O(m^2)$, and there are several sub-quadratic approximations available [4] A

[3] In comparative genomics, exchanging two elements is sometimes called a *super short transposition*.

related edit distance is the *LCS* distance, which is the minimum number of character insertions or deletions needed to transform one set of symbols into another.

Like the *LCS* distance, sorting by move operations can be interpreted as finding a minimal sequence of edit operations where the only operations allowed are insertions and deletions. If these operations each take uniform cost and the universe of strings is restricted to *permutations*, the corresponding edit distance has a special name called the *permutation edit distance*. We obtain the following lower bound size of a sorting (i.e., a schedule) using moves.

PROPOSITION 3.2 (Schedule size). *Given a pair of filtrations* (K_0, K_1) *and a one-to-one correspondence* $f : K_0 \to K_1$*, the coarsest move schedule* S^* *is of size* $\Omega(d)$*, where:*

$$d = m - |\mathrm{LCS}(K_0, K_1)|$$

Moreover, any size d *schedule* S *can be computed in* $O(m \log\log m)$ *time.*

Proof. Recall our definition of edit distance given above, depending on the choice of $S \subseteq S_m$ of allowable edit operations, and that in order for any edit distance to be symmetric, if $s \in S$ then $s^{-1} \in S$. This implies that $d_S(p, q) = d_S(p^{-1}, q)$ for any choice of $p, q \in S_m$. Moreover, edit distances are *left-invariant*, i.e.

$$d_S(p, q) = d_S(r \circ p, r \circ q) \quad \text{for all } p, q, r \in S_m$$

Conceptually, left-invariance implies that the edit distance between any pair of permutations p, q is invariant under an arbitrary relabeling p, q–as long as the relabeling is consistent. Thus, the following identity always holds:

$$d_S(p, q) = d_S(\iota, p^{-1} \circ q) = d_S(q^{-1} \circ p, \iota)$$

where $\iota = [m]$, the identity permutation. Suppose we are given permutations $p = K_0, q = K_1$, and we seek to compute $\mathrm{LCS}(p, q)$. Consider the permutation $r = p^{-1} \circ q$. Since the LCS distance is a valid edit distance, if $|\mathrm{LCS}(p, q)| = k$, then $|\mathrm{LCS}(\iota, r)| = k$ as well. Notice that ι is strictly increasing–thus any common subsequence ι has with r must also be strictly increasing. Thus, the problem of computing $\mathrm{LCS}(p, q)$ reduces to the problem of computing the *longest increasing subsequence* (LIS) of r, which can done in $O(m \log\log m)$ time [7]. The optimality of d follows from the optimality of the well-studied LCS problem [31]. ∎

COROLLARY 3.3. *If* K_0, K_1 *are random distinct filtrations each of size* m*, then* d *is upper bounded by* $O(m - \sqrt{m})$ *in expectation, with probability* 1 *as* $m \to \infty$*.*

Proof. The proof of this result reduces to showing the average length of the LIS for random permutations. Let $L(p) \in [1, m]$ denote the maximal length of a increasing subsequence of $p \in S_m$. The essential quantity to show the expected length of $L(p)$ over all permutations:

$$\ell_m = \mathbb{E}\, L(p) = \frac{1}{m!} \sum_{p \in S_m} L(p)$$

The history of the mathematics of estimating this quantity dates back at least 50 years, and is sometimes called the *Ulam-Hammersley* problem. The seminal work by Baik et al. [3] established that as $m \to \infty$:

$$\ell_m = 2\sqrt{m} + cm^{1/6} + o(m^{1/6})$$

where $c = -1.77108....$ As a result, we have:

$$\frac{\ell_m}{\sqrt{m}} \to 2 \quad \text{as} \quad m \to \infty$$

Thus, if $p \in S_m$ denotes a uniformly random permutation in S_m, then $L(p)/\sqrt{m} \to 2$ in probability as $m \to \infty$. Since the schedule size is defined as $d = m - |\text{LCS}(K_0, K_1)|$, the claimed bound follows. ∎

The upper bound from Corollary 3.3 captures the size of S^* under worst-case conditions. It is worth noting that this bound applies to pairs of uniformly sampled permutations, as opposed to uniformly sampled distinct filtrations—intuitively, simplicial filtrations have more structure due to the subcomplex relation that must be respected in the filtration. Thus, while $O(m - \sqrt{m})$ in expectation is an improvement over $O(m)$, it is still not tight. However, Boissonnat [9] has shown that for any d-dimensional simplicial complex K with m simplices, the number of distinct filtrations built from K is *at least* $\lfloor \frac{t+1}{d+1} \rfloor^m$, where t is the number of distinct filtration values associated with the complex K. Since this lower bound grows similarly to $m!$ when $t \sim O(m)$ and $d << m$ fixed, we expect $O(m - \sqrt{m})$ is not too pessimistic of an upper bound when the pair of filtrations are truly random. In practice, when one has a time-varying filtration and the sampling points are relatively close [in time], the LCS between adjacent filtrations is expected to be much larger, subsequently making d much smaller. For example, the average size of the LCS across the 10 filtrations from Section 1.2 each having $m = 417$ simplices sampled at evenly spaced time points was 343, implying $d \approx 70$ permutations needed on average to update the decomposition between adjacent time points.

3.3. Minimizing schedule cost.

3.3.1. Motivation. The results from the previous section yield a sufficient algorithm for generating move schedules of minimal cardinality: simply compute a $\text{LCS}(K_0, K_1)$ for a pair (K_0, K_1) and fix an order of simplices in the set $\text{LCS}(K_0, K_1) \setminus K_0$ to move. Any sequence of moves which increases the LCS on each move is guaranteed to have size $m - |\text{LCS}(K_0, K_1)|$, and the reduction to the permutation edit distance problem ensures this size is optimal. However, like in the vineyards algorithm, certain pairs of simplices cost more to exchange depending on whether they are critical pairs in the sense described [21], making specific move operations more expensive than others.

One advantage of Algorithm 2.2 is that the cost of applying a permutation to some given $R = DV$ decomposition can be determined efficiently prior to performing any column operations. Specifically, the cost of determining $|\mathbb{I}| + |\mathbb{J}|$ in lines (2) and (3) of Algorithm 2.2 cannot take more than $O(m)$ time. If we assume a similar complexity to determine the cost of applying a permutation with Algorithm 2.3 and we know to execute d moves, an initial thought might be to use an optimization method like dynamic programming. That is, suppose $O(dm)$ move costs are computed ahead of time and these costs are stored in a table providing $O(1)$ element access and modification. Using a sparse row representation for V, this cannot take more than $O(dm)$ time, which is relatively efficient when $d << m$. With this structure, determining the cost of the cheapest move takes just $O(d)$ time. However, moving a simplex from position i to position j may require updating $O(\kappa^2)$ entries in the structure, where $\kappa = |i - j|$. Since $\kappa \sim O(m)$ in the worst case, direct optimization may require upwards of $\approx O(dm^2)$. Since this time complexity is potentially more

expensive than executing a schedule to begin with, this direct strategy does not seem viable for practical settings.

3.3.2. Greedy approach. If an optimization procedure like dynamic programming is too expensive, one would hope that a greedy solution which chooses the minimal cost choice at every step could lead be an efficient strategy. We give a counter-example below demonstrating that a greedy procedure may lead to arbitrarily bad behavior.

Counter-Example A pair of filtrations is given below, each filtration contains $m = 10$ simplices comprising the 1-skeletons of a 3-simplex with a particular choice of ordering. Relabeling the simplices of K_0 to the index set $[m]$ and modifying the labels of K_1 correspondingly yields permutations given below:

$$K_0 = \{a\ b\ c\ d\ u\ v\ w\ x\ y\ z\} = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$$

$$K_1 = \{a\ b\ c\ d\ x\ y\ z\ u\ v\ w\} = 1\ 2\ 3\ 4\ 8\ 9\ 10\ 5\ 6\ 7$$

The subset of the filtration which corresponds to the simplices which lie in the LCS between these permutations is colored in red. For this example, the permutation edit distance is $d = m - \mathrm{LCS}(K_0, K_1)$ implies exactly 3 moves are needed to apply the mapping $K_0 \mapsto K_1$. There are six possible valid schedules of moves:

$$\begin{array}{lll} S_1 = m_{xu}, m_{yu}, m_{zu} & S_3 = m_{yu}, m_{xy}, m_{zu} & S_5 = m_{zu}, m_{xz}, m_{yz} \\ S_2 = m_{xu}, m_{zu}, m_{yz} & S_4 = m_{yu}, m_{zu}, m_{xy} & S_6 = m_{zu}, m_{yz}, m_{xz} \end{array}$$

The cost of each successive move operation for each of these schedules is recorded in Table 3.1. A greedy strategy which always selects the cheapest move in succession

Table 3.1: Move schedule costs

Cost of each permutation				
	1st	2nd	3rd	Total
S_1	2	3	1	6
S_2	2	2	4	8
S_3	4	2	2	8
S_4	4	3	3	10
S_5	2	2	4	8
S_6	2	5	3	10

would begin by moving x or z first, since these are the cheapest moves available, which implies one of S_1, S_2, S_5, S_6 would be picked depending on the choice of a tie-breaker. While the cheapest schedule S_1 is in this candidate set, so is S_6, the most expensive schedule. Moreover, a tie-breaker that keeps track of previous cheapest moves would end up picking either schedule S_2 or S_5 as the final greedy schedule, both of which are suboptimal cost-wise.

Nonetheless, greedy strategies can be effective in practice, even if they are suboptimal. In the previous example, since evaluating the cost of move is $\approx O(m)$, a greedy strategy that chooses the least expensive move would require $O(d^2 m)$ time, which is computationally viable when $d << m$.

3.3.3. Proxy objective. In light of the direct optimization being too expensive and a greedy procedure leading to suboptimal behavior, we consider whether there exists an alternative, or *proxy objective* to optimize that is related to the cost of a schedule but not dependent on the entries in the decomposition. In this section, we present such an objective, showing its relation to the problem at hand as well its use in other known computational problems. We show empirical results demonstrating its practical effectiveness as a heuristic in section 4.

As before, given two permutations $p, q \in S_m$, let $S = (s_1, s_2, \ldots, s_d)$ denote a sorting such that $S \circ p = q$. In general, there are exponentially many ways to generate such a sorting S. The example given in section 3.3.2 shows all (minimal size) $d!$ possible choices of S for a fixed pair of permutations and choice of $\mathrm{LCS}(p, q)$. Let $\tau = \mathrm{LCS}(p, q)$ and let $\sigma \in S_d$ denote a choice of ordering for the symbols in $p \setminus \tau$, such that $S(\sigma)$ denotes a schedule generated from this ordering. Ideally, we would like to achieve something similar to the following objective:

$$S^*_\sigma = \arg\min_{\sigma \in S_d} \mathrm{cost}(S(\sigma) \circ p) \tag{3.6}$$

where $\mathrm{cost}(S(\sigma) \circ p)$ measures the number of column operations require to execute $S(\sigma)$ on an existing decomposition. The schedule S_1 shown in Table 3.1 would be an example of a minimal cost schedule. Since optimizing this objective combinatorially is inefficient, we would like an alternative objective that is cheaper and performs well in practice. Intuitively, one alternative objective might be the Kendall-τ distance K_τ, which can be computed in $O(m \log m)$ time—indeed, this is the weighting measure minimized in a certain path optimization in the RIVET software [35] to ensure adjacent filtrations are similar. Motivated by this, define $\hat{s}_i = s_i \circ \cdots \circ s_2 \circ s_1 \circ p$ to be the result of applying the composition of the first i permutations for some given schedule $S = (s_1, \ldots, s_d)$, where $i < d$. A possible inequality to utilize is the following:

$$K_\tau(p, q) = K_\tau(p, S \circ p) \leq \sum_{i=1}^{d-1} K_\tau(\hat{s}_i, \hat{s}_{i+1}) \tag{3.7}$$

When the set of permutations S are adjacent transpositions, equality is achieved above since $K_\tau(\hat{s}_i, \hat{s}_{i+1}) = 1$ for every inversion between p and q. However, this is not guaranteed if S was derived using the strategy from section 3. Obtaining S^*_σ via direct optimization over S_d isn't practical, but experimental tests suggest that a greedy-type procedure performs well as a heuristic, which yields a $O(d^2 m \log m)$ minimization procedure. When $d << m$, this strategy of successively minimizing the Kendall-τ distance is efficient.

However, when $d \sim O(m)$, the quadratic nature of this minimization is too expensive. It turns out there is a similar distance that is both cheaper to compute and also amenable to a greedy type optimization. The *Spearman distance* F is a common distance for measuring the disarrangement between permutations. It is defined as:

$$F(p, q) = \sum_{i \in [m]} |p(i) - q(i)| \tag{3.8}$$

Intuitively, the Spearman distance can be interpreted as ℓ_1-type distance between two permutations. Like the Kendall-τ distance, it is a metric, it is invariant under relabeling, and the distance between pairs of uniformly random permutations $p, q \sim$

S_m are asymptotically normal. Indeed, Diaconis et al. [25] showed the following inequality relating the Spearman distance and the Kendall-τ distances:

$$K_\tau(p,q) \le F(p,q) \le 2K_\tau(p,q) \tag{3.9}$$

Although very similarity to the Kendall-τ distance, the Spearman distance offers some convenient computational advantages: it can be computed in $O(m)$ time, and the variance of its asymptotic distribution is larger. Consider the following minimization:

$$S_\sigma^* = \arg\min_{\sigma \in S_d} \frac{1}{2} \sum_{i=1}^{d-1} F(\hat{s}_i, \hat{s}_{i+1}) \tag{3.10}$$

The minimizer of this optimization can be interpreted as a sorting which minimizes the net displacement amongst all possible sortings, again in the ℓ_1 sense. Given two permutations $p, q \in S_m$ and a LCS $\tau = \text{LCS}(p,q)$ between them, the analysis in section 3 shows we must move each symbol $x \in \mathcal{D}$ where $\mathcal{D} = p \setminus \tau$ in a way that extends τ after each move. Thus we have $d = |\mathcal{D}|$ symbols to choose from initially. Each symbol $x \in \mathcal{D}$ yields a permutation s to apply to p. Unlike the Kendall distance, the Spearman distance satisfies:

$$F(p,q) \ge F(s_1 \circ p, q) \ge F(s_2 \circ s_1 \circ p, q) \ge F(s_d \circ \cdots \circ s_2 \circ s_1 \circ p, q) = F(q,q) = 0$$

For any sorting S such that s_i that increases the size of longest common subsequence between $s_{i-1} \circ \cdots \circ s_1 \circ p$ and q. Since $|\mathcal{D}|$ is fixed, we would like to choose $x \in \mathcal{D}$ such that the corresponding permutation s_x reduces the total displacement of every symbol $x \in \mathcal{D}$ relative to q. A greedy strategy chooses x such that $F(\hat{s}_i, \hat{s}_{i+1})$ is as small as possible, removes that symbol from $\mathcal{D}$, and recurses. Thus, a greedy strategy depends on our ability to quickly compute $F(\hat{s}_i, \hat{s}_{i+1})$.

Example: An example of three possible schedules, S_1, S_2, and S_3 is given in Figure 3.1. Each schedule, from left to right, has 21, 31, and 35 crossings, respectively. Each column represents the successive application of permutations from the three fixed schedules. Black/red vertices correspond to symbols in and outside of $\text{LCS}(p,q)$, respectively. All three schedules were generated from the same $\text{LCS}(p,q) = (4\,7\,8)$ and

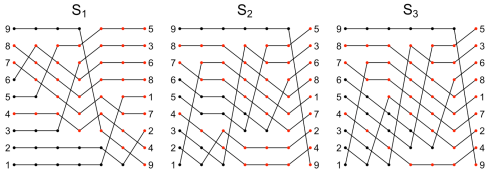


Figure 3.1: Example of three possible schedules, S_1, S_2, and S_3.

all three schedules transform $p \mapsto q$ in $d = 6$ moves. The greedy heuristic proposed to minimize equation 3.10 was used to produce S_1, which has the minimal number of crossings amongst all possible schedules in this example, since $K_\tau(p,q) = 21$. This minimization problem is related to a constrained version of the bipartite crossing minimization problem for k sets of permutations, which is NP-hard for $k \ge 4$ [8].

The monotonic nature of successively applying the Spearman distance enables certain

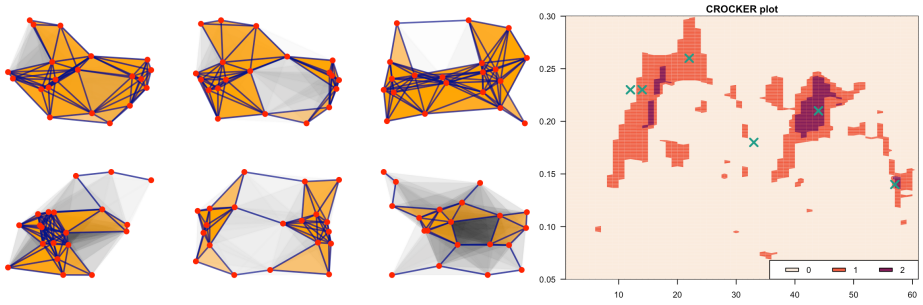


Figure 4.1: An example of a crocker plot (right) depicting the evolution of dimension $p = 1$ Betti curves over time. The green X marks correspond chronologically to the complexes (left), in row-major order. The large orange and purple areas depict 1-cycles persisting in both space (the y-axis) and time (x-axis).

computational advantages. Suppose we begin with an array $\mathcal{A}$ of size m which provides $O(1)$ access and modification, initialized to 0. Summing the entries in this array yields the Spearman distance in $O(m)$ time, which at initialization represents $F(p, p)$. Each symbol $x \in \mathcal{D}$ induces a permutation s_x such that $F(p, q) \geq F(s_x \circ p, q)$. Since there are $d = |\mathcal{D}|$ symbols to move and computing F takes $O(m)$ time, a naïve greedy strategy requires $O((d + (d-1) + (d-2) + \cdots + 1)m) \approx O(d^2 m)$ time to complete. However, observe that each permutation s_x changes the displacement of every symbol in a very predictable way: if s_x represents a move permutation that moves x from position i to position j, then relative to the previous permutation s_i, the displacement of x in s_{i+1} changes by at most $|i - j|$ and the displacement of any symbol between i and j changes by ± 1. All other symbols are unaffected. Since the Spearman distance is simply the sum of these displacements, replacing $\mathcal{A}$ with a structure that supports both $O(\log m)$ access time to aggregate information (such as the sum) and $O(\log m)$ modification time ability to up to $|i - j|$ entries. The former problem reduces to the problem of efficiently calculating and updating *prefix sums*, however since $|i - j| \leq m$ is potentially larger than $O(\log m)$, it's not immediately clear how to achieve the latter modification complexity in $O(\log m)$ time. Fortunately, since every elements displacement other than x changes by ± 1 per update, both queries can be efficiently done in $O(\log m)$ time using a Segment Tree that supports *lazy propagation* [22].

4. Empirical results.

Crocker stacks. There are many challenges to depicting topological behavior in dynamic settings. One approach is to trace out the curves constituting a continuous family of persistence diagrams in $\mathbb{R}^3$—the vineyards approach—however this visualization can be cumbersome to work with as there are potentially many such vines tangled together, making topological critical events with low persistence difficult to detect. Moreover, the vineyards visualization does not admit a natural simplification utilizing the stability properties of persistence, as individual vines are not stable: if two vines move near each other and then pull apart without touching, then a pairing in their corresponding persistence diagrams may cross under a small perturbation, signaling the presence of an erroneous topological critical event [44, 46].

Acknowledging this, Topaz et al. [44] proposed the use of a 2-dimensional sum-

mary visualization, called a *crocker*[4] plot. In brief, a crocker plot is a contour plot on a family of Betti curves. Formally, given a filtration $K = K_0 \subseteq K_1 \subseteq \cdots \subseteq K_m$, a p-dimensional *Betti curve* $\beta_p^\bullet$ is defined as the ordered sequence of p-th dimensional Betti numbers:

$$\beta_p^\bullet = \{\, \mathrm{rank}(H_p(K_0)),\, \mathrm{rank}(H_p(K_1)),\, \ldots,\, \mathrm{rank}(H_p(K_m)) \,\}$$

Given a time-varying filtration $K(\tau)$, a crocker plot can be interpreted as a contour plot on the 1-parameter family of Betti curves $\beta_p^\bullet(\tau)$. A example of a crocker plot generated from the simulation described below is given in Figure 4.1. Since only the Betti numbers at each simplex in the filtration are needed to generate these Betti curves, the persistence diagram is not directly needed to generate a crocker plot; it is sufficient to use e.g. any of the specialized methods discussed in 1.3. This dependence only on the Betti numbers makes crocker plots easier to compute than standard persistence, however what one gains in efficiency one loses in stability; it is known that Betti curves are inherently unstable with respect to small fluctuations about the diagonal of the persistence diagram.

Acknowledging this, Xian et al. [46] showed that crocker plots may be *smoothed* to inherit the stability property of persistence diagrams and reduce noise in the visualization. That is, when applied to a time-varying persistence module $M = \{M_t\}_{t\in[0,T]}$ an α-smoothed crocker plot for $\alpha \geq 0$ is the rank of the map $M_t(\epsilon-\alpha) \to M_t(\epsilon+\alpha)$ at time t and scale ϵ. For example, the standard crock plot is a 0-smoothed crocker plot. Allowing all three parameters (t, ϵ, α) to vary continuously leads to 3D visualization called an *α-smoothed crocker stack*.

Definition 4.1 (crocker stack). *A crocker stack is a family of α-smoothed crock plots which summarizes the topological information of a time-varying persistence module M via the function $f_M : [0,T] \times [0,\infty) \times [0,\infty) \to \mathbb{N}$, where:*

$$f_M(t, \epsilon, \alpha) = \mathrm{rank}(M_t(\epsilon - \alpha) \to M_t(\epsilon + \alpha))$$

A crocker stack is a sequence of α-smoothed crocker plots that vary over $\alpha \geq 0$, satisfying $f_M(t, \epsilon, \alpha) \leq f_M(t, \epsilon, \alpha)$ for all $\alpha \geq \alpha'$. Note that, unlike crocker plots, applying this α smoothing efficiently *requires* the persistence pairing. Indeed, it has been shown that crocker stacks and stacked persistence diagrams (i.e. vineyards) are equivalent to each other in the sense that either one contains the information needed to reconstruct the other [46]. Thus, computing crocker stacks reduces to computing the persistence of a (time-varying) family of filtrations.

We test the efficiency of computing the necessary information to generate these crocker stacks using a spatiotemporal data set to illustrate the applicability of our method. Specifically, we ran a *flocking* simulation similar to the simulation run in [44] with $m = 20$ vertices moving around on the unit square equipped with periodic boundary conditions (i.e. $S^1 \times S^1$). We simulated movement by equipping the vertices with a simple set of rules which control how the individual vertices position change over time. Such simulations are also called *boid* simulations, and they have been extensively used as models to describe how the evolution of collective behavior over time can be described by simple sets of rules. The simulation is initialized with every vertex positioned randomly in the space; the positions of vertices over time is

[4] *crocker* stands for "Contour Realization Of Computed k-dimensional hole Evolution in the Rips complex." Although the acronym includes *Rips complexes* in the name, in principle a crocker plot could just as easily be created using other types of triangulations (e.g. Čech filtrations).

updated according to a set of rules related to the vertices acceleration, distance to other vertices, etc. To get a sense of the time domain, we ran the simulation until a vertex made at least 5 rotations around the torus.

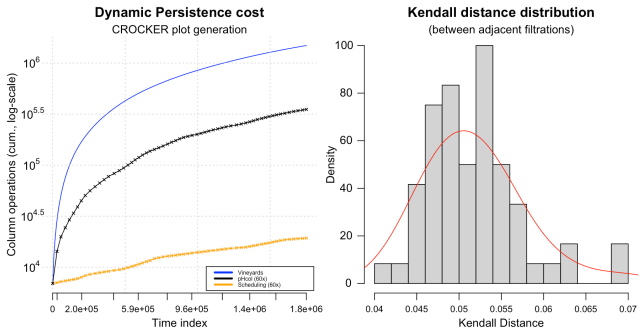


Figure 4.2: The cumulative number of $O(m)$ column operations (left, log-scale) of the three approaches tested and the distribution of the normalized Kendall distance between adjacent filtration pairs. The histogram on the right depicts the coarseness of the discretization; most filtration pairs have $\approx 5\%$ of their $\approx O(m^2)$ simplex pairs inverted between adjacent time steps.

Given this time-evolving data set, we computed the persistence diagram of the Rips filtration up to $\epsilon = 0.30$ at 60 evenly spaced time points using three approaches: the standard algorithm `pHcol` applied naïvely at each of the 60 time steps, the vineyards algorithm applied to (linear) homotopy connecting filtrations adjacent in time, and our approach using moves. The cumulative number of $O(m)$ column operations executed by three different approaches. Note again that vineyards requires generating many decompositions by design (in this case, $\approx 1.8M$). The standard algorithm `pHcol` and our move strategy were computed at 60 evenly spaced time points of the simulations. As depicted in Figure 4.2, our move strategy is far more efficient then both vineyards and the naive `pHcol` strategies.

Multidimensional persistence. A natural question is whether or not there exists a multidimensional generalization of persistence. Given a structure which filters the space in multiple dimensions simultaneously, a *multifiltration*, the goal of multidimensional persistence is to identify persistent features by examining the entire multifiltration. Such a generalization has appeared naturally in many application contexts, showing potential as a tool for exploratory data analysis [37]. For example, one of the often noted drawbacks of persistence is its instability with respect to strong outliers. Although persistence diagrams are stable with respect to their input and thus are robust to noise, the presence of strong outliers can artificially modify the connectivity of the underlying filtration, obscuring the detection of what would otherwise be significant topological structures [12]. Indeed, since inferring the homology of some underlying topological space around which data are distributed was one of the original purposes of persistence[5], strong outliers pose a significant barrier to the widespread adoption of persistent homology. By filtering the space according to e.g.

[5]One of the first uses of persistence was to infer the topology of attractors in dynamical systems from experimental data[41].

some notion of density, multidimensional persistence enables the practitioner to handle outliers directly, without resorting to sophisticated data preprocessing techniques which may or may not preserve the underlying topology of the data.

Unfortunately, multidimensional persistent homology comes with its own set of computational and theoretical challenges not necessarily inherited by the 1-D case. There is not complete discrete invariant for multidimensional persistence, and the computations associated to other incomplete invariants exhibit high algorithmic complexity. A number of computational strategies have been proposed to counter this, such as homology preserving multifiltration simplification or parallelization schemes using shared-memory computation [27]. Rather than attempting to tackle multiple dimensions simultaneously, another approach is to focus on a fixed dimension, such as 2-D persistence. Utilizing the equivalence between the rank and fibered barcode invariants, Lesnick and Wright [35] developed an elegant way of visualizing the former via a reparameterization of the latter using standard point-line duality. This clever reparameterization effectively reduces the fibered barcode computation to a sequence of 1-D barcode computations at "template points" lying within the 2-cells of particular planar subdivision of the half-plane $[0, \infty) \times \mathbb{R}$. Indeed, since algorithm 3.1 was designed for precisely such a computation, the algorithm proposed by [35] is prototypical of the class of methods that stand to benefit from move scheduling.

The purpose of the following sections is to uncover the intrinsic bottlenecks impeding the computation of the fibered barcode invariant in practical settings. Towards this goal, we briefly summarize the theory established by several researchers [35, 15, 33], focusing on their computational and algorithmic details. Our goal is to show that the highest complexity subcomputation of the fibered barcode invariant algorithm proposed by [35] is precisely the type of computation we focus on in this effort. In doing so, we substantiate our claims from section 1.2 that our scheduling approach can improve the efficiency of a real-world application of multidimensional persistence by an order of magnitude by comparing Algorithm 3.1 to several alternatives on a real-world data set and problem: the problem of identifying the underlying topological space of a data set extracting from high-contrast natural images.

The Fibered Barcode. Lesnick and Wright [35] associate three simple invariants to a 2-dimensional persistence module M: the *dimension function*, the *multigraded Betti numbers*, and the *fibered barcode*. The dimension function is simply the the function which maps every point $a \in \mathbb{R}^2$ to $\dim(M_a)$. It is a simple and easy to visualize invariant, but is unstable and yields no information about the persistent features of M. The multigraded Betti numbers constitute a natural and important class of invariants often studied in algebraic geometry commutative algebra, though both their theory and computation is beyond the scope of this effort—the interested reader is referred to [35, 16] for more details. For our purposes, the i^{th}-graded Betti number of M is simply a function $\beta_i(M) : \mathbb{R}^n \to \mathbb{N}$ whose values $\beta_i^z(M)$ indicate the the number of elements at grade z in a basis of the i^{th} module in a free resolution for M. The fibered barcode, denoted as $\mathcal{B}(M)$, is defined as follows:

Definition 4.2 (Fibered barcode). *The fibered barcode $\mathcal{B}(M)$ of a bipersistence module M is the map which sends each line L with non-negative slope to the barcode $\mathcal{B}_L(M)$:*

$$\mathcal{B}(M) = \{\ B_L(M) : L \in \mathbb{R} \times \mathbb{R}^+\ \}$$

Thus, given a bipersistence module M, the fibered barcode of M is the 2-parameter family of barcodes given by restricting M to the of set affine lines with non-negative slope in $\mathbb{R}^2$.

Although the fiber barcode is a very intuitive invariant, it is not clear how one might go about computing it efficiently. One could compute the 1-D persistence associated to some fixed linear combination of two given filter functions; each linear combination parameterizes a line $L \in \mathbb{R} \times \mathbb{R}^+$ which M restricts too. However, this an $O(m^3)$ time computation, which is prohibitive for exploratory data analysis purposes. Ideally, one would want to continuously vary $L \in \mathbb{R} \times \mathbb{R}^+$ interactively, inspecting the corresponding changes to the barcodes immediately. To enable this, [35] introduced an elegant reparameterization that not only fully characterizes $\mathcal{B}(M)$, but also yields an efficient method for rendering $\mathcal{B}_L(M)$ in real time. In the next section, we briefly summarize this reparameterization.

Reparameterization using point-line duality. Let $\overline{\mathcal{L}}$ denote the collection of all lines in $\mathbb{R}^2$ with non-negative slope, $\mathcal{L} \subset \overline{\mathcal{L}}$ the collection of all lines in $\mathbb{R}^2$ with non-negative, finite slope, and $\mathcal{L}^\circ$ the collection of all affine lines in $\mathbb{R}^2$ with positive, finite slope. There is a standard point-line duality that gives a parameterization of $\mathcal{L}$ with the half-plane $[0, \infty) \times \mathbb{R}$ that is convenient to work with here. Define the *line* and *point* dual transforms $\mathcal{D}_\ell$ and $\mathcal{D}_p$, respectively, as follows:

$$\begin{aligned} \mathcal{D}_\ell : \mathcal{L} &\to [0,\infty) \times \mathbb{R} & \mathcal{D}_p : [0,\infty) \times \mathbb{R} &\to \mathcal{L} \\ y = ax + b &\mapsto (a, -b) & (c, d) &\mapsto y = cx - d \end{aligned} \tag{4.1}$$

The transforms $\mathcal{D}_\ell$ and $\mathcal{D}_p$ are *dual* to each other in the sense that for any point $a \in [0, \infty) \times \mathbb{R}$ and any line $L \in \mathcal{L}$, $a \in L$ if and only if $D_\ell(L) \in D_p(a)$. Now, for some fixed line L, define the *push map* $\text{push}_L(a) : \mathbb{R}^2 \to L \cup \infty$ as:

$$\text{push}_L(a) \mapsto \min\{v \in L \mid a \leq v\} \tag{4.2}$$

The push map satisfies a number of useful properties. Namely:

1 For $r < s \in \mathbb{R}^2$, $\text{push}_L(r) \leq \text{push}_L(s)$
2 For each $a \in \mathbb{R}^2$, $\text{push}_L(a)$ is continuous on $\mathcal{L}^\circ$
3 For $L \in \mathcal{L}^\circ$ and $S \subset \mathbb{R}^2$, push_L induces a totally ordered partition S_L on S

Property (1) asserts that for any $L \in \overline{\mathcal{L}}$, the standard partial order on $\mathbb{R}^2$ restricts to a total order on L. Properties (2) and (3) qualify the following definition:

Definition 4.3 (Critical Lines). *For some fixed $S \subset \mathbb{R}^2$, a line $L \in L^\circ$ is defined to be* regular *if there is an open ball $B \in L^\circ$ containing L such that $S_L = S_{L'}$ for all $L' \in B$. Otherwise, the line L is defined as* critical.

The set of critical lines $\text{crit}(M)$ with respect to some fixed set $S \subset \mathbb{R}^2$ fully characterizes a certain planar subdivision of the half plane $[0, \infty) \times \mathbb{R}$. Explicitly, define $\mathcal{A}^1(M)$ to the 1-skeleton of the line arrangement defined by:

$$\mathcal{A}^1(M) = \{\mathcal{D}_\ell(\text{crit}(M)) \cup (\{0\} \times \mathbb{R})\} \tag{4.3}$$

Expanding the 1-skeleton to a 2-D cell complex $\mathcal{A}(M)$ fully partitions the upper half plane. A consequence of equation 4.3 that if the duals of two lines $L, L' \in \mathcal{L}$ are contained in the same 2-cell in $\mathcal{A}(M)$, then $S_L = S_{L'}$, i.e. the partitions induced by push_L are equivalent. Indeed, the total order on S_L is the pullback of the total order on L with respect to the push map. Since $\mathcal{A}(M)$ partitions the entire half-plane, the dual to every line $L \in \mathcal{L}$ is contained within $\mathcal{A}(M)$—the desired reparameterization.

Let $S = \text{supp}\, \beta_0(M) \cup \text{supp}\, \beta_1(M)$, where the functions $\beta_0(M), \beta_1(M)$ are 0^{th} and 1^{st} bigraded Betti numbers of M, respectively. The main mathematical result from [35] is a characterization of the barcodes $\mathcal{B}_L(M)$, for any $L \in \mathcal{L}$, in terms of

a set of *barcode templates* $\mathcal{T}$ computed at every 2-cell in $\mathcal{A}(M)$. More formally, for any line $L \in \overline{\mathcal{L}}$ and e any 2-cell in $\mathcal{A}(M)$ whose closure contains the dual of L under point-line duality, the 1-parameter restriction of the persistence module M induced by L is given by:

$$\mathcal{B}_L(M) = \{[\,\mathrm{push}_L(a),\, \mathrm{push}_L(b)\,) \mid (a,b) \in \mathcal{T}^e, \mathrm{push}_L(a) < \mathrm{push}_L(b)\} \tag{4.4}$$

Minor additional conditions are needed for handling completely horizontal and vertical lines. The importance of this theorem lies in the fact that the fibered barcodes are completely defined from the precomputed barcode templates $\mathcal{T}$—once every barcode template $\mathcal{T}^e$ has been computed and augmented onto $\mathcal{A}(M)$, $\mathcal{B}(M)$ is completely characterized, and the barcodes $\mathcal{B}_L(M)$ associated to a 1-D filtration induced by *any* choice of L can be efficiently computed via a point-location query on $\mathcal{A}(M)$ and a $O(|\mathcal{B}_L(M)|)$ application of the push map.

Computing the fibered barcode. Given bipersistence module M, the three dominant computations needed to compute the fibered barcode using the reparameteratization discussed above are the following:

1 Computing the bigraded Betti numbers $\beta(M)$ of M
2 Constructing a line arrangement $\mathcal{A}(M)$ induced by critical lines from (1)
3 Augmenting $\mathcal{A}(M)$ with a *barcode template* $\mathcal{T}^e$ at every 2-cell $e \in \mathcal{A}(M)$

Computing (1) takes approximately $\approx O(m^3)$ using a matrix algorithm similar to Algorithm 2.1 [36]. Constructing and storing the line arrangement $\mathcal{A}(M)$ with n lines and k vertices is related to the *line segment intersection problem*, which known algorithms in computational geometry can solve in (optimal) output-sensitive $O((n+k)\log n)$ time (see [10] for an overview). In terms of memory complexity, the number of 2-cells in $\mathcal{A}(M)$ is upper bounded by $O(\kappa^2)$, where κ is a coarseness parameter associated with the computation of $\beta(M)$. By combining this bound with a bound on the number of transpositions needed to traverse a particular sequence of 2-cells in $\mathcal{A}(M)$, the analysis from [35] shows that the computation of the barcodes templates $\mathcal{T}$ from step (3) using vineyard updates requires on the order of $O(m^3\kappa + m\kappa^2 \log \kappa)$ elementary operations and $O(m\kappa^2)$ storage.

Of the three steps above, the barcode template computation is both the highest complexity and most demanding computation in practice. The number of 2-cells in $\mathcal{A}(M)$ is on the order $O(\kappa^2)$, and κ itself is on the order of $O(m^2)$ in the worst case. This suggests the computation of the barcode templates is on the order $O(m^5)$, which is effectively intractable for even moderately-sized filtrations. In practice, the external stability result from [33] justifies the use of a simple procedure which approximates the module M with a smaller module M' using a grid-like coarsening procedure. Two binning parameters associated with the coarsening procedure, x_{bin} and y_{bin}, together determine the degree of the approximation. The coarsening procedure allows the practitioner to restrict the size of κ to a relatively small constant, which in-turn dramatically reduces the size of $\mathcal{A}(M)$.

There are several approaches one can use to compute $\mathcal{T}$, the simplest being to run Algorithm 2.1 independently on the 1-D filtration induced by the duals of some set of points (e.g. the barycenters) lying in the interior of the 2-cells of $\mathcal{A}(M)$. The approach taken by [35] is to use the $R = DV$ decomposition computed at some adjacent 2-cell $e \in \mathcal{A}(M)$ to speed up the computation of an adjacent cell $e' \in \mathcal{A}(M)$. More explicitly, define the *dual graph* of $\mathcal{A}(M)$ to be the undirected graph G which has a vertex for every 2-cell $e \in \mathcal{A}(M)$ and an edge for each adjacent pair of cells $e, e' \in \mathcal{A}(M)$. Each vertex in G is associated with a barcode template $\mathcal{T}^e$, and the

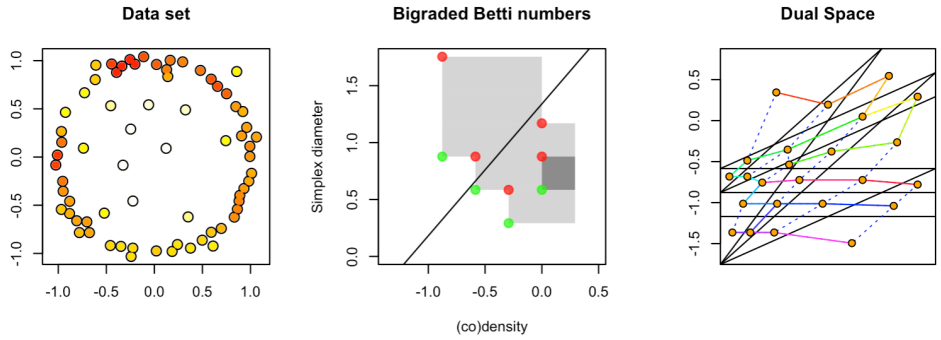


Figure 4.3: Bipersistence example on an 8×8 coarsened grid. On left, the input data, colored by density. In the middle, the bigraded Betti numbers $\beta_0(M)$ and $\beta_1(M)$ (green and red, respectively), the Hilbert function (gray), and a line L emphasizing the persistence of features with high density. On the right, the line arrangement $\mathcal{A}(M)$ lying in the dual space $D_p(\alpha)$ derived from the $\beta(M)$.

computation of $\mathcal{T}$ now reduces to computing a path Γ on G which visits each vertex at least once. To minimize the computation time, assume the n edges of G are endowed with non-negative weights $W = w_1, w_2, \ldots, w_n$ whose values $w_i \in \mathbb{R}_+$ represent some notion of distance. The optimal path Γ^* is the one of minimal length with respect to W which visits every vertex of G at least once. There is a known $\frac{3}{2}$-approximation that can be computed efficiently which reduces the problem to the traveling salesman problem on a metric graph [19].

Noisy circle example We include a small example to illustrate many of the concepts and definitions discussed so far. Consider a small set of points distributed noisily around S^1 that contains a few strong outliers, shown on the left side of Figure 4.3. Filtering this data set with respect to the Rips parameter and the complement of a kernel density estimate yields a bifiltration whose dimension function and bigraded Betti numbers are shown in the middle figure. The gray areas indicate the region where the dimension function is constant; the lighter gray area where $\dim_1(M) = 1$, indicating where a persistent loop was detected. On the right side, the space dual to the space in the middle is shown. The black lines represent the lines of $\mathcal{A}(M)$, the blue dashed-lines the edges of the dual graph $\mathcal{G}$, and the orange points the barycenters of the 2-cells in $\mathcal{A}(M)$, depicting where the barcodes templates $\mathcal{T}^e$ are parameterized. The shortest path Γ^* spanning the vertices of G is shown by the solid rainbow colored lines, with the traversal beginning in the top-left cell (at the red edge) and ending at the bottom-right cell (the indigo edge). Traversing Γ^* can be done via a simple depth-first search, though note that a $R = DV$ decomposition needs to be stored at each degree-3 or higher vertex in Γ^* to avoid redundant computations as the traversal recurses.

Natural images dataset. In the previous section, we outlined the computational theory of multidimensional persistence, including its relevance to our proposed move scheduling approach. In this section, we demonstrate the efficiency of our method on practical use case of multidimensional persistence: detecting the pres-

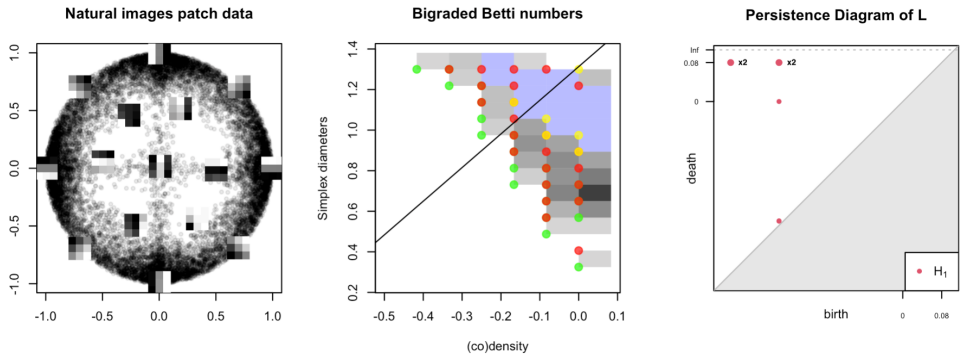


Figure 4.4: Bipersistence example of natural images data set on an 12×16 coarsened grid. On the left, a projection the full data set is shown, along with the 15 landmark patches. In the middle, the bigraded Betti numbers, the Hilbert function, and a chosen line L are shown for the coarsened grid. As before, the $0/1/2$ dimension bigraded Betti numbers are shown in green/red/yellow, respectively. The blue regions highlights where $\dim(M) = 5$. On the right, the persistence diagram of M restricted to L, with larger points indicating a multiplicity of 2. Observe five persistent features are revealed in both the middle and right plots, matching β_1 of the three-circle model.

ence of a low-dimensional topological space which well-approximates the distribution of texture patches extracted from real-world image data.

A common hypothesis is that high dimensional data tend to lie in the vicinity of an embedded, low dimensional manifold or topological space. An exemplary demonstration of this is given in [34], who explored the space of 3×3 high-contrast patches extracted from Hans van Hateren's still image collection [28], which consists of approximately 4,000 monochrome images[6] depicting various areas outside around Groningen (Holland). Motivated by applications to sparse coding applications, Lee et al. [34] was interested in how these natural image patches were distributed, in pixel-space, with respect to predicted spaces and manifolds. For example, one of the motivating questions of their research was whether there were any clear qualitative differences the distributions of patches extracted from from images of different modalities, e.g. optical versus range images. A major result from their work is that the majority of high-contrast image patches are concentrated around a 2-dimensional submanifold embedded in S^7. Supplemental experiments conducted by Carllsson et al. [14] using persistent homology confirmed that the distribution of high-contrast 3×3 patches is indeed well-approximated by a Klein bottle—around 60% of the high-contrast patches from the still image data set are concentrated within a small neighborhood around a model of a Klein bottle which accounts for $\approx 21\%$ of the volume S^7, compared to the $\approx 84\%$ of volume of S^7 accounted for by the entire data set. Along a similar vein, in the context of sparse coding, Perea et al. [42] introduced a dictionary learning framework for estimating and representing the distribution of patches from texture images.

If one was not aware of the analysis done by [34, 28, 14, 42], it is not immediately clear a priori that the Klein bottle model is indeed a good candidate for capturing non-

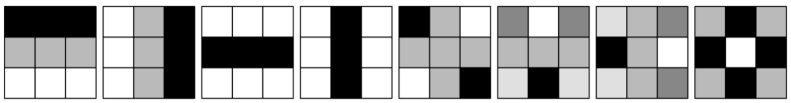
[6] See http://bethgelab.org/datasets/vanhateren/ for details on the image collection.

linearity of image patches. Nonetheless, as we have mentioned, among the first uses of persistent homology was the so-called "homology inference problem" [41]; given a suitable treatment of strong outliers using by e.g. filtering both on the geometry and the density of the image patches, the homology of the Klein bottle should become apparent through a persistence computation.

Consider the (coarsened) fibered barcode computed from a standard Rips / co-density bifiltration on a representative sample of the image data from [28] (details to follow), shown in Figure 4.4. Upon inspection of the bigraded Betti number and the dimension function, one finds that a large area of dimension function is constant (highlighted as the blue portion in the middle of Figure 4.4), wherein the first Betti number is 5. There are many spaces whose meets this criterion, though upon further inspection, one candidate space that seems particularly plausible is the three-circle model C_3. The C_3 model consists of three circles, two of which (say, S_v and S_h) intersect the third (say, S_{lin}) in exactly two points, but themselves do not intersect. Upon performing an analysis on the density of the patches analogous to the one done by [34], one finds that the mean-centered and contrast normalized 3×3 patches like on a 7-dimensional ellipsoid $\tilde{S}^7 \subset \mathbb{R}^9$. Contrast is typically measured using a discrete version of the Dirichlet semi-norm, or the "D-norm" $\|\cdot\|_D$ for short, which is a unique scale-invariant norm on images:

$$\|x\|_D = \sqrt{\sum_{i \sim j} (x_i - x_j)^2} = \sqrt{x^T D x}$$

where D is a fixed matrix which upon application to an image $x \in \mathbb{R}^9$ yields a value proportional to the sum of the differences between the 4-connected neighbors (described by the relation $i \sim j$). It is advantageous to "whiten" the data via a change of coordinates to Euclidean sphere. This can be achieved by expressing the patch data as an expansion of a certain convenient basis of 8 non-constant eigenvectors, often called the Discrete Cosine Transform (DCT) basis [34], which are given below:



Projecting the image data onto the first two basis vectors of leads to the projection shown in the top left of Figure 4.4. 15 landmark points are also shown. Observe the data are distributed well around three "circles"—the outside circle capturing the rotation gradient of the image patches (S_{lin}), and the other two circles capturing the vertical and horizontal gradients (S_v and S_h, respectively). Since the three circle model is the 1-skeleton of the Klein bottle [14], one may conclude that the Klein bottle might be a reasonable candidate upon which the image data are distributed. This can be further verified empirically by e.g. using the iterative, meanshift-like procedure described in [14] to fit a parameterization of the model in question to the data.

Using the same high-contrast patch data set studied in [34], we evaluate the performance of various methods at computing the fibered barcode invariant using the parameterization from 4. Due to the aforementioned high complexity of the fibered barcode computation, we begin by working with a subset of the image patch data $\mathcal{X}$. Specifically, we combine the use furthest-point sampling and proportionate allocation (stratified) sampling to sample landmarks $X \subset \mathcal{X}$ distributed within $n = 25$ strata.

Each strata consists of the $(1/n)$-thick level set given by ρ_{15}, the density estimator. The use of furthest-point sampling gives us certain coverage guarantees that the geometry is approximately preserved within each level set, whereas the stratification ensures the original density of is approximated preserved as well. From this data set, we construct a Rips-(co)density bifiltration using ρ_{15} equipped with the geodesic metric computed over the same $k = 15$ knn graph on X. Finally, we record the number of column reductions needed to compute the fibered barcode at a variety of levels of coarsening using `pHcol`, vineyards, and our moves approach. The results are summarized in Table 4.1. We also record the number of 2-cells encountered along the path Γ permutations that were applied throughout the computation of the barcode templates for both vineyards and moves, denoted in the table as d_K and d_{LCS}, respectively.

Table 4.1: Cost to computing $\mathcal{T}$ for various coarsening choices of $\beta(M)$.

$\beta(M)$	$\mathcal{A}(M)$	Column reductions / Permutations		
Coarsening	# 2-cells	`phCol`	Vineyards / d_K	Moves / d_{LCS}
8 x 8	39	94.9K	245K / 1.53M	38.0K / 11.6K
12 x 12	127	318K	439K / 2.66M	81.9K / 33.0K
16 x 16	425	1.07M	825K / 4.75M	114K / 87.4K
20 x 20	926	2.32M	1.15M / 6.77M	148K / 154K
24 x 24	1.53K	3.92M	1.50M / 8.70M	184K / 232K

As shown on the table, we're able to achieve a significant reduction in the number of total column operations needed to compute $\mathcal{T}$ compared to both vineyards and `pHcol`, though it's worth noting that permutations are constant time for vineyards while they require at most $O(m)$ elementary operations for moves. Observe that initially, when there is a high degree of coarsening, vineyards is particularly inefficient and moves requires only about 3x less column operations that naively computing $\mathcal{T}$ independently. As the coarsening becomes more refined and more 2-cells are added to $\mathcal{A}(M)$, however, vineyards quickly becomes a much more viable option compared to `pHcol`, though even at the highest coarsening we tested the gain in efficient was relatively small. In contrast, our proposed moves approach scales quite well with this refinement, requiring about 12% and $\approx 5\%$ of the number of column operations as vineyards and pHcol, respectively.

5. Conclusion and Future Work. In conclusion, we presented a scheduling algorithm for efficiently updating a decomposition in sparse dynamic settings. Our approach is simple, easy to implement, and fully general: it does not depend on the geometry of underlying space nor the choice of triangulation. Moreover, we supplied efficient algorithms for our scheduling strategy, provided tight bounds where applicable, and demonstrated our algorithms performance with several real world use cases.

There are many possible applications of our work beyond the example application of generating crocker stacks and simulating multidimensional persistence. As discussed in sections 1.3, examples include:

1 Computing persistent images for time-varying systems
2 Optimization procedures involving persistence diagrams
3 Detecting homological critical points in time-varying filtrations

Indeed, we envision our approach as potentially useful to any situation where the structure of interest is a parameterized family of filtrations and the goal involves computing the persistent homology for each filtration in the family. Areas of particular

interest include time-series analysis and dynamic metric spaces [30].

The simple and combinatorial nature of approach does pose some limitations to its applicability. There may be better bounds or algorithms available if stronger assumptions can be made on how the filtration is changing. If the filtration K_0 shares little similarity to the "target" filtration K_1, then the overhead of reducing the simplices from $K_1 \setminus K_0$ appended to the decomposition derived from K_0 may be large enough to motivate simply computing the decomposition at K_1 independently, especially if parallel processors are available. This overhead does not exist for the multidimensional persistence use-case: the entire decomposition is needed at each template 2-cell.

From an implementation perspective, one non-trivial complication of our approach is its heavy dependence on a particular sparse matrix data structure which permits permuting both the row and columns of a given matrix in at most $O(m)$ time. Our use of this data structure is motivated by the fact that, like vineyards and unlike the standard persistence algorithm, the $R = DV$ decomposition is permuted with every $O(m)$ operation. Indeed, as shown with the natural images example in section 4, there are often more permutation operations being applied than there are column reductions. In the more standard *compressed* sparse matrix representations (i.e. CSC, CSR, etc.), permuting both the rows and columns generally takes at most $O(Z)$ time, where Z is the number of non-zero entries. If the particular filtration has many cycles, permuting these compressed representations can be quite expensive, bottlenecking the computation both in theory and in practice. As a result, the more complex sparse matrix representation from [21] is necessary to be efficient in practice.

Moving forward, our results suggest there are many aspects of computing persistence in dynamic settings yet to be explored. For example, it's not immediately clear whether one could adopt, for example, the twist optimization used in the reduction algorithm to the dynamic setting. Another avenue of interest is the setting where one has knowledge a priori of the time-varying... particular spaces

REFERENCES

[1] H. Adams, T. Emerson, M. Kirby, R. Neville, C. Peterson, P. Shipman, S. Chepushtanova, E. Hanson, F. Motta, and L. Ziegelmeier, *Persistence images: A stable vector representation of persistent homology*, Journal of Machine Learning Research, 18 (2017).

[2] D. Attali, M. Glisse, S. Hornus, F. Lazarus, and D. Morozov, *Persistence-sensitive simplication of functions on surfaces in linear time*, in TopoInVis' 09, 2009.

[3] J. Baik, P. Deift, and K. Johansson, *On the distribution of the length of the longest increasing subsequence of random permutations*, Journal of the American Mathematical Society, 12 (1999), pp. 1119–1178.

[4] Z. Bar-Yossef, T. Jayram, R. Krauthgamer, and R. Kumar, *Approximating edit distance efficiently*, in 45th Annual IEEE Symposium on Foundations of Computer Science, IEEE, 2004, pp. 550–559.

[5] U. Bauer, *Ripser: efficient computation of vietoris–rips persistence barcodes*, Journal of Applied and Computational Topology, (2021), pp. 1–33.

[6] U. Bauer, M. Kerber, J. Reininghaus, and H. Wagner, *Phat–persistent homology algorithms toolbox*, Journal of symbolic computation, 78 (2017), pp. 76–90.

[7] S. Bespamyatnikh and M. Segal, *Enumerating longest increasing subsequences and patience sorting*, Information Processing Letters, 76 (2000), pp. 7–11.

[8] T. Biedl, F. J. Brandenburg, and X. Deng, *On the complexity of crossings in permutations*, Discrete Mathematics, 309 (2009), pp. 1813–1823.

[9] J.-D. Boissonnat and K. CS, *An efficient representation for filtrations of simplicial complexes*, ACM Transactions on Algorithms (TALG), 14 (2018), pp. 1–21.

[10] J.-D. Boissonnat and F. P. Preparata, *Robust plane sweep for intersecting segments*, SIAM Journal on Computing, 29 (2000), pp. 1401–1421.

[11] J.-D. Boissonnat and J. Snoeyink, *Efficient algorithms for line and curve segment intersection using restricted predicates*, Computational Geometry, 16 (2000), pp. 35–52.
[12] M. Buchet, F. Chazal, T. K. Dey, F. Fan, S. Y. Oudot, and Y. Wang, *Topological analysis of scalar fields with outliers*, in 31st International Symposium on Computational Geometry, Schloss Dagstuhl, Leibniz-Zentrum fü Informatik GmbH, 2015, pp. 827–841.
[13] O. Busaryev, T. K. Dey, and Y. Wang, *Tracking a generator by persistence*, Discrete Mathematics, Algorithms and Applications, 2 (2010), pp. 539–552.
[14] G. Carlsson, T. Ishkhanov, V. De Silva, and A. Zomorodian, *On the local behavior of spaces of natural images*, International journal of computer vision, 76 (2008), pp. 1–12.
[15] G. Carlsson, G. Singh, and A. J. Zomorodian, *Computing multidimensional persistence*, Journal of Computational Geometry, 1 (2010), pp. 72–100.
[16] G. Carlsson and A. Zomorodian, *The theory of multidimensional persistence*, Discrete & Computational Geometry, 42 (2009), pp. 71–93.
[17] C. Chen and M. Kerber, *Persistent homology computation with a twist*, in Proceedings 27th European Workshop on Computational Geometry, vol. 11, 2011, pp. 197–200.
[18] C. Chen and M. Kerber, *An output-sensitive algorithm for persistent homology*, Computational Geometry, 46 (2013), pp. 435–447.
[19] N. Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, tech. report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.
[20] D. Cohen-Steiner, H. Edelsbrunner, and J. Harer, *Stability of persistence diagrams*, Discrete & computational geometry, 37 (2007), pp. 103–120.
[21] D. Cohen-Steiner, H. Edelsbrunner, and D. Morozov, *Vines and vineyards by updating persistence in linear time*, in Proceedings of the twenty-second annual symposium on Computational geometry, 2006, pp. 119–126.
[22] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry*, in Computational geometry, Springer, 1997, pp. 1–17.
[23] V. De Silva, D. Morozov, and M. Vejdemo-Johansson, *Dualities in persistent (co) homology*, Inverse Problems, 27 (2011), p. 124003.
[24] C. J. A. Delfinado and H. Edelsbrunner, *An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere*, Computer Aided Geometric Design, 12 (1995), pp. 771–784.
[25] P. Diaconis and R. L. Graham, *Spearman's footrule as a measure of disarray*, Journal of the Royal Statistical Society: Series B (Methodological), 39 (1977), pp. 262–268.
[26] H. Edelsbrunner, D. Letscher, and A. Zomorodian, *Topological persistence and simplification*, in Proceedings 41st annual symposium on foundations of computer science, IEEE, 2000, pp. 454–463.
[27] U. Fugacci and M. Kerber, *Chunk reduction for multi-parameter persistent homology*, in 35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA, 2019, pp. 37–1.
[28] J. H. v. Hateren and A. v. d. Schaaf, *Independent component filters of natural images compared with simple cells in primary visual cortex*, Proceedings: Biological Sciences, 265 (1998), pp. 359–366.
[29] B. M. Kapron, V. King, and B. Mountjoy, *Dynamic graph connectivity in polylogarithmic worst case time*, in Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms, SIAM, 2013, pp. 1131–1142.
[30] W. Kim and F. Mémoli, *Spatiotemporal persistent homology for dynamic metric spaces*, Discrete & Computational Geometry, (2020), pp. 1–45.
[31] S. K. Kumar and C. P. Rangan, *A linear space algorithm for the lcs problem*, Acta Informatica, 24 (1987), pp. 353–362.
[32] A. Labarre, *Lower bounding edit distances between permutations*, SIAM Journal on Discrete Mathematics, 27 (2013), pp. 1410–1428.
[33] C. Landi, *The rank invariant stability via interleavings*, arXiv preprint arXiv:1412.3374, (2014).
[34] A. B. Lee, K. S. Pedersen, and D. Mumford, *The nonlinear statistics of high-contrast patches in natural images*, International Journal of Computer Vision, 54 (2003), pp. 83–103.
[35] M. Lesnick and M. Wright, *Interactive visualization of 2-d persistence modules*, arXiv preprint arXiv:1512.00180, (2015).
[36] M. Lesnick and M. Wright, *Computing minimal presentations and bigraded betti numbers of 2-parameter persistent homology*, arXiv preprint arXiv:1902.05708, (2019).
[37] M. P. Lesnick, *Multidimensional interleavings and applications to topological inference*, Stan-

ford University, 2012.
[38] D. Morozov, *Persistence algorithm takes cubic time in worst case*, BioGeometry News, Dept. Comput. Sci., Duke Univ, 2 (2005).
[39] P. Oesterling, C. Heine, G. H. Weber, D. Morozov, and G. Scheuermann, *Computing and visualizing time-varying merge trees for high-dimensional data*, in Topological Methods in Data Analysis and Visualization, Springer, 2015, pp. 87–101.
[40] N. Otter, M. A. Porter, U. Tillmann, P. Grindrod, and H. A. Harrington, *A roadmap for the computation of persistent homology*, EPJ Data Science, 6 (2017), pp. 1–38.
[41] J. A. Perea, *A brief history of persistence*, arXiv preprint arXiv:1809.03624, (2018).
[42] J. A. Perea and G. Carlsson, *A klein-bottle-based dictionary for texture representation*, International journal of computer vision, 107 (2014), pp. 75–97.
[43] L. Polanco and J. A. Perea, *Adaptive template systems: Data-driven feature selection for learning with persistence diagrams*, in 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), IEEE, 2019, pp. 1115–1121.
[44] C. M. Topaz, L. Ziegelmeier, and T. Halverson, *Topological data analysis of biological aggregation models*, PloS one, 10 (2015), p. e0126383.
[45] M. Ulmer, L. Ziegelmeier, and C. M. Topaz, *A topological approach to selecting models of biological experiments*, PloS one, 14 (2019), p. e0213679.
[46] L. Xian, H. Adams, C. M. Topaz, and L. Ziegelmeier, *Capturing dynamics of time-varying data via topology*, arXiv preprint arXiv:2010.05780, (2020).
[47] A. Zomorodian and G. Carlsson, *Computing persistent homology*, Discrete & Computational Geometry, 33 (2005), pp. 249–274.

Appendix.

Vineyard Algorithm.

Transposition complexity details. Consider Algorithm .1. The matrices R, V are both $(m \times m)$, where m is the number of simplices in the underlying filtration. Each transposition $\mathrm{Tr}(i, i+1)$ requires at most 2 column operations (denoted with S_i^j in the algorithm) in both R and V, a constant number of queries to the matrix representation, and exactly 1 row and column exchange. Column operations take $O(m)$ time. Querying to determine whether an entry at position $(i, i+1)$ is non-zero depends on the sparse matrix implementation, but is generally either $O(\log m)$ or $O(m)$. In compressed sparse matrix representation, column and row exchanges take upwards of $O(z)$ time, where z is the number of non-zero entries in the matrix, however this can be made $O(1)$ time by using the data structure described in [21]. This data structure requires storing an index using at most $O(m \log m)$ bits with each non-zero entry, as well as several $O(m)$-sized auxiliary arrays to protect the matrices from row exchanges.

Algorithm .1 Vineyards: Tranposition Framework

Require: Matrices R, V satisfying $R = DV$, $1 \leq i \leq m-1$

Ensure: Output (R, V) maintains the decomposition invariants

1: **function** TRANSPOSE(R, V, i)
2: pos $\leftarrow$ columns satisfying $\mathrm{col}_R = 0$
3: **if** pos[i] **and** pos[$i+1$] **then**
4: **if** $V[i, i+1] \neq 0$ **then**
5: $\mathrm{col}_V(i+1)$ += $\mathrm{col}_V(i)$
6: **if** $\exists\, k, l$ satisfying $\mathrm{low}_R(k) = i, \mathrm{low}_R(l) = i+1$ and $R[i,l] \neq 0$ **then** ▷ O(m)
7: **if** $k < l$ **then**
8: **return** (R, V) $\leftarrow$ ($PRPS^l_k$, $PVPS^l_k$)
9: **else**
10: **return** (R, V) $\leftarrow$ ($PRPS^k_l$, $PVPS^k_l$)
11: **else if** !pos[i] **and** !pos[$i+1$] **then**
12: **if** $V[i, i+1] \neq 0$ **then** ▷ O(m)
13: **if** $\mathrm{low}_R(i) < \mathrm{low}_R(i+1)$ **then**
14: **return** (R, V) $\leftarrow$ ($PRS^{i+1}_i P$, $PVS^{i+1}_i P$)
15: **else**
16: **return** (R, V) $\leftarrow$ ($PRS^{i+1}_i PS^{i+1}_i$, $PVS^{i+1}_i PS^{i+1}_i$)
17: **else if** !pos[i] **and** pos[$i+1$] **then**
18: **if** $V[i, i+1] \neq 0$ **then** ▷ O(m)
19: **return** (R, V) $\leftarrow$ ($PRS^{i+1}_i PS^{i+1}_i$, $PVS^{i+1}_i PS^{i+1}_i$)
20: **else if** pos[i] **and** !pos[$i+1$] **then**
21: **if** $V[i, i+1] \neq 0$ **then** ▷ O(m)
22: $\mathrm{col}_V(i+1)$ += $\mathrm{col}_V(i)$
23: **return** (R, V) $\leftarrow$ (PRP, PVP)