Move Schedules: Fast persistence computations in coarse dynamic settings *

Matthew Piekenbrock[1] and Jose A. Perea[2]

[1] Khoury College of Computer Sciences, Northeastern University.
[2] Department of Mathematics and Khoury College of Computer Sciences, Northeastern University.

Contributing authors: piekenbrock.m@northeastern.edu; j.pereabenitez@northeastern.edu;

Abstract

Matrix reduction is the standard procedure for computing the persistent homology of a filtered simplicial complex with $\boldsymbol{m}$ simplices. Its output is a particular decomposition of the total boundary matrix, from which the persistence diagrams and generating cycles are derived. Persistence diagrams are known to vary continuously with respect to their input, motivating the study of their computation for time-varying filtered complexes. Computing persistence dynamically can be reduced to maintaining a valid decomposition under adjacent transpositions in the filtration order. Since there are $\boldsymbol{O(m^2)}$ such transpositions, this maintenance procedure exhibits limited scalability and often is too fine for many applications. We propose a coarser strategy for maintaining the decomposition over a 1-parameter family of filtrations. By reduction to a particular longest common subsequence problem, we show that the minimal number of decomposition updates $\boldsymbol{d}$ can be found in $\boldsymbol{O(m \log\log m)}$ time and $\boldsymbol{O(m)}$ space, and that the corresponding sequence of permutations—which we call a *schedule*—can be constructed in $\boldsymbol{O(dm \log m)}$ time. We also show that, in expectation, the storage needed to employ this strategy is actually sublinear in $\boldsymbol{m}$. Exploiting this connection, we show experimentally that the decrease in operations 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. Applications to video data, dynamic metric space data, and multi-parameter persistence are also presented.

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

1 Introduction

Given a triangulable topological space equipped with a 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 if the function changes continuously, so too will the points on the persistence diagram [1, 2]. This motivates the application of persistence to time-varying settings, like that of dynamic metric spaces [3]. As persistence-related computations tend to exhibit high algorithmic complexity—essentially cubic[1] in the size of the underlying filtration [4]—their adoption to dynamic settings poses a challenging computational problem. Currently, there is no recourse when faced with a time-varying complex containing millions of simplices across thousands of snapshots in time. Acquiring such a capability has far-reaching consequences: methods that vectorize persistence diagrams for machine learning purposes all immediately become computationally viable tools in the dynamic setting. Such persistence summaries include adaptive template functions [5], persistence images [6], and α-smoothed Betti curves [7].

Cohen-Steiner et al. refer to a continuous 1-parameter family of persistence diagrams as a *vineyard*, and they give in [2] an efficient algorithm for their computation. The vineyards approach can be interpreted as an extension of the *reduction* algorithm [8], which computes the persistence diagrams of a filtered simplicial complex K with m simplices in $O(m^3)$ time, via a particular decomposition $R = DV$ (or $RU = D$) of the boundary matrix D of K. The vineyards algorithm, in turn, transforms a time-varying filtration into a certain set of permutations of the decomposition $R = DV$, each of which takes 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. Moreover, the vineyards algorithm is efficient asymptotically: if there are d time-points where the filtration order changes, then *vineyards* takes $O(m^3 + md)$ time; one initial $O(m^3)$-time reduction at time t_0 followed by one $O(m)$ operation to update the decomposition at the remaining time points $(t_1, t_2, \ldots, t_d)$. When $d >> m$, the initial reduction cost is amortized by the cost of maintaining the decomposition, implying each diagram produced takes just linear time per time point to obtain.

Despite its theoretical efficiency, *vineyards* is often not the method of choice in practical 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 [9] for an overview), implementations of the vineyards algorithm are relatively uncommon.[2] The reason for this disparity is perhaps explained by Lesnick and Wright [10]: "While an update to an *RU*

[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.

[2] Dionysus 1 does have an implementation of *vineyards*, however the algorithm was never ported to version 2. Other major packages, such as GUDHI and PHAT, do not have *vineyards* implementations.

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, they observe that maintaining the decomposition along a certain parameterized family is the most computationally demanding aspect of RIVET [11], a software for computing and visualizing two-parameter persistent homology.

The work presented here seeks to further understand and remedy this discrepancy: building on the work presented in [12], we introduce a coarser approach to the vineyards algorithm. Though the vineyards algorithm is efficient at constructing a *continuous* 1-parameter family of diagrams, it is not necessarily efficient when the parameter is coarsely discretized. Our methodology is based on the observation that practitioners often don't need (or want!) all of the persistence diagrams generated by a continuous 1-parameter of filtrations; usually just $n << d$ of them suffice. By exploiting the "donor" concept introduced in [12], we are able to make a tradeoff between the number of times the decomposition is restored to a valid state and the granularity of the decomposition repair step, reducing the total number of column operations needed to apply an arbitrary permutation to the filtration. This trade off, paired with a fast greedy heuristic explained in section 3.4.2, yields an algorithm that can update a $R = DV$ decomposition more efficiently than *vineyards* in coarse time-varying contexts, making dynamic persistence more computationally tractable for a wider class of use-cases. The source code containing both the algorithm we propose and the experiments performed in Section 4 is open source and available online.[3]

1.1 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 [2] and the moves algorithm [12], both of which are discussed extensively in section 2. At the time of writing, we were made aware of very recent work [13] that iteratively repairs a permuted decomposition via a column swapping and reduce strategy, which they call "warm starts." Though their motivation is similar to our own, their approach relies on the reduction algorithm as a subprocedure, which is quite different from the strategy we employ here.

Contrasting the dynamic setting, there is extensive work on improving the efficiency of computing a single (static) $R = DV$ decomposition. Chen [14] 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 [15], which dramatically improves the effectiveness of the clearing optimization [16]. There are many other optimizations on the implementation side: the use of ranking functions defined on the

[3]For all accompanying software and materials, see: https://github.com/peekxc/move_schedules

combinatorial number system enables implicit cofacet enumeration, removing the need to store the boundary matrix explicitly; the apparent/emergent pairs optimization identifies columns whose pivot entries are unaffected by the reduction algorithm, reducing the total number of columns which need to 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 [16, 17].

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 algorithms. For example, Attali et al. [18] give a linear time algorithm 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. [19] give a fast incremental algorithm for computing persistent Betti numbers up to dimension 2, again by utilizing symmetry, duality, and "time-reversal" [20]. Chen & Kerber [21] 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-cubic rank algorithms specialized for sparse-matrices.

If zeroth homology is the only dimension of interest, computing and updating both the persistence and rank information is greatly simplified. For example, if the edges of the graph are in filtered order 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. [22] give an algorithm that maintains a *merge tree* with e edges in $O(e)$ time per-update. If only Betti numbers are needed, the zeroth-dimension problem reduces even further to the *dynamic connectivity problem*, which can be be efficiently solved in amortized $O(\log n)$ query and update times using either Link-cut trees or multi-level Euler tour trees [23].

1.2 A Motivating Example

To motivate this effort, we begin with an illustrative example of why the vineyards algorithm does not always yield an efficient strategy for time-varying settings. Consider a series of grayscale images (i.e. a video) depicting a fixed-width annulus expanding about the center of a 9×9 grid, and its associated sublevel-set filtrations, as shown in Figure 1.

Each image in the series is comprised of pixels whose intensities vary with time, upon which we build a simplicial complex using the *Freudenthal* triangulation of the plane. For each complex, we create a filtration of simplices whose order is determined by the lower stars of pixel values. Two events critically change the persistence diagrams: the first occurs when the central connected component splits to form a cycle, and the second when

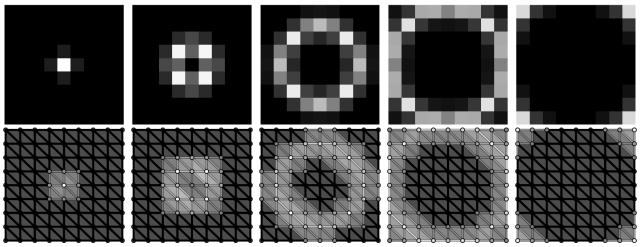


Figure 1: Top: A video of an expanding annulus. Bottom: Sublevel-set filtrations, via (negative) pixel intensity, of a Freudenthal triangulation of the plane.

the annulus splits into four components. From left to right, the Betti numbers of the five evenly spaced 'snapshots' of the filtration shown above are: $(\beta_0, \beta_1) = (1, 0),\ (1, 1),\ (1, 1),\ (1, 1),\ (4, 0)$. Thus, in this example, only a few persistence diagrams are needed to capture the major changes to the topology.

We use this data set as a baseline for comparing *vineyards* and the standard reduction algorithm `pHcol` (Algorithm 4). Suppose a practitioner wanted to know the major homological changes a time-varying filtration encounters over time. Since it is unknown a priori when the persistent pairing function changes, one solution is to do n independent persistence computations at n evenly spaced points in the time domain. An alternative approach is to construct a homotopy between a pair of filtrations (K, f), (K, f') and then decompose this homotopy into adjacent transpositions based on the filtration order—the vineyards approach. We refer to the former as the *discrete setting*, which is often used in practice, and the latter as the *continuous setting*. Note that though the discrete setting is often more practical, it is not guaranteed to capture all homological changes in persistence that occur in the continuous 1-parameter family of diagrams.

The cumulative cost (in total column operations) of these various approaches are shown in Figure 2, wherein the reduction (`pHcol`) and vineyard algorithms are compared. Two discrete strategies (green and purple) and two continuous strategies (black and blue) are shown.

Note that without knowing where the persistence pairing function changes, a continuous strategy must construct all $\approx 7 \times 10^4$ diagrams induced by the homotopy. In this setting, as shown in the figure, 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 coarse enough, the naive approach actually performs less column operations than *vineyards*, 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 framework must incur some overhead (in terms of column operations) to maintain the underlying decomposition, even when the pairing function determining the persistence diagram is unchanged. Indeed, as shown by the case where $n = 10$, applying `pHcol` independently between relatively "close"

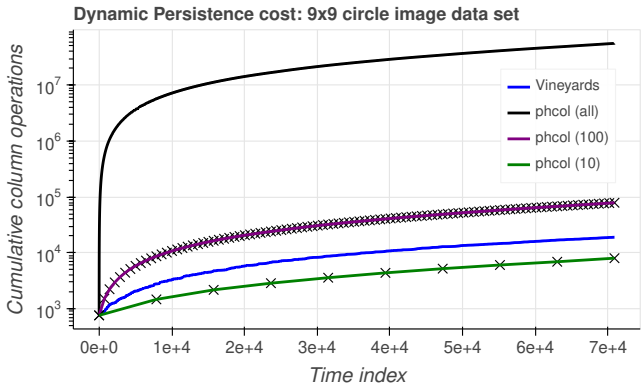


Figure 2: The cumulative column operations needed to compute persistence across the time-varying filtration of grayscale images. Observe 10 independent persistence computations evenly spaced in time (green line) captures the major topological changes and is the most computationally efficient approach shown.

filtrations is substantially more efficient than iteratively updating the decomposition. Moreover, any optimizations to the reduction algorithm (e.g. clearing [14]) would only increase this disparity. Since persistence has found many applications in dynamic contexts [3, 10, 24, 25], a more efficient alternative to *vineyards* is clearly needed.

Our approach and contributions are as follows: First, we leverage the *moves* framework of Busaryev et al. [12] to include coarser operations for dynamic persistence settings. By a reduction to an edit distance problem, we give a lower bound on the minimal number of moves needed to perform an arbitrary permutation to the $R = DV$ decomposition, along with a proof of its optimality. We also give worst-case sizes of these quantities in expectation as well as efficient algorithms for constructing these operations—both of which are derived from a reduction to the Longest Increasing Subsequence (LIS) problem. These operations parameterize sequences of permutations $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ of minimal size d, which we call *schedules*. However, not all minimal size schedules incur the same cost (i.e., number of column operations). We investigate the feasibility of choosing optimal cost schedules, and show that greedy-type approaches can lead to arbitrarily bad behavior. In light of these results, we give an alternative proxy-objective for cost minimization, provide bounds justifying its relevance to the original problem, and give an efficient $O(d^2 \log m)$ algorithm for heuristically solving this proxy minimization. A performance comparison with other reduction-based persistence computations is given, wherein move schedules are demonstrated to be an order of magnitude more efficient than existing approaches at calculating persistence in dynamic settings. In particular, we illustrate the effectiveness of efficient scheduling with a variety of real-world applications, including flock

analysis in dynamic metric spaces and manifold detection from image data using 2D persistence computations.

1.3 Main results

Given a simplicial complex K with filtration function f, denote by $R = DV$ the decomposition of its corresponding boundary matrix D such that R is reduced and V is upper-triangular (see section 2.1 for details). If one has a pair of filtrations (K, f), (K, f') of size $m = |K|$ and $R = DV$ has been computed for (K, f), then it may be advantageous to use the information stored in (R, V) to reduce the computation of $R' = D'V'$. Given a permutation P such that $D' = PDP^T$, such an update scheme has the form:

$$(*P * R * P^T*) = (PDP^T)(*P * V * P^T*)$$

where $*$ is substituted with elementary column operations that repair the permuted decomposition. It is known how to linearly interpolate $f \mapsto f'$ using $d \sim O(m^2)$ updates to the decomposition, where each update requires at most two column operations [2]. Since each column operation takes $O(m)$, the complexity of reindexing $f \mapsto f'$ is $O(m^3)$, which is efficient if all d decompositions are needed. Otherwise, if only (R', V') is needed, updating $R \mapsto R'$ using the approach from [2] matches the complexity of computing $R' = D'V'$ independently.

We now summarize our main results (Theorem 1): suppose one has a schedule $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ yielding a corresponding sequence of decompositions:

$$R = R_0 = D_0V_0 \xrightarrow{s_1} D_1V_1 \xrightarrow{s_2} \ldots \xrightarrow{s_d} D_dV_d = R_d = R' \tag{1}$$

where $s_k = (i_k, j_k)$ for $k = 1, \ldots, d$, denotes a particular type of cyclic permutation (see section 3.2). If $i_k < j_k$ for all $s_k \in \mathcal{S}$, our first result extends [12] by showing that (1) can be computed using $O(\nu)$ column operations, where:

$$\nu = \sum_{k=1}^{d} |\mathbb{I}_k| + |\mathbb{J}_k| \tag{2}$$

The quantities $|\mathbb{I}_k|$ and $|\mathbb{J}_k|$ depend on the sparsity of the V_k and R_k matrices, respectively, and $d \sim O(m)$ is a constant that depends on how similar f and f' are. The advantage of this result is that it depends explicitly on the sparsity pattern of the decomposition itself and is thus output sensitive, which we leverage in Section 3.4.

Our second result turns towards lower bounding $d = |\mathcal{S}|$ and the complexity of constructing $\mathcal{S}$ itself. By reinterpreting a special set of cyclic permutations as edit operations on strings, we find that a *smallest* such sequence mapping

f to f', has size (Proposition 3) :

$$d = m - |\mathrm{LCS}(f, f')| \tag{3}$$

where $\mathrm{LCS}(f, f')$ refers to the size of the longest common subsequence between the simplexwise filtrations (K, f) and (K, f') (see section 3.2 for more details). We also show that the information needed to construct any $\mathcal{S}$ with optimal size can be computed in $O(m \log \log m)$ preprocessing time and $O(m)$ memory. We provide evidence that $d \sim m - \sqrt{m}$ in expectation for random filtrations (Corollary 2). Although this implies d can be $O(m)$ for pathological inputs, we give empirical results suggesting d can be much smaller in practice.

Outline: The paper is organized as follows: we review and establish the notations we will use to describe simplicial complexes, persistent homology, and dynamic persistence in Section 2. We also cover the reduction algorithm (designated here as `pHcol`), the vineyards algorithm, and the set of *move*-related algorithms introduced in [12], which serves as the starting point of this work. In Section 3 we introduce move schedules and provide efficient algorithms to construct them. In Section 4 we present applications of the proposed method, including the computation of crocker stacks from flock simulations and of a 2-dimensional persistence invariant on a data set of image patches derived from natural images. In Section 5 we conclude the paper by discussing other possible applications and future work.

2 Background

Suppose one has a family $\{K_i\}_{i \in I}$ of simplicial complexes indexed by a totally ordered set I, and so that for any $i < j \in I$ we have $K_i \subseteq K_j$. Such a family is called a *filtration*, which is deemed *simplexwise* if $K_j \setminus K_i = \{\sigma_j\}$ whenever j is the immediate successor of i in I. Any finite filtration may be trivially converted into a simplexwise filtration via a set of *condensing*, *refining*, and *reindexing* maps (see [16] for more details). Equivalently, a filtration can be also defined as a pair (K, f) where K is a simplicial complex and $f : K \to I$ is a *filter function* satisfying $f(\tau) \leq f(\sigma)$ in I, whenever $\tau \subseteq \sigma$ in K. In this setting, $K_i = \{\, \sigma \in K : f(\sigma) \leq i \,\}$. Here, we consider two index sets: $[m] = \{1, \ldots, m\}$ and $\mathbb{R}$. Without loss of generality, we exclusively consider simplexwise filtrations, but for brevity-sake refer to them simply as filtrations.

Let K be an abstract simplicial complex and $\mathbb{F}$ a field. A p-chain is a formal $\mathbb{F}$-linear combination of p-simplices of K. The collection of p-chains under addition yields an $\mathbb{F}$-vector space denoted $C_p(K)$. The p-boundary $\partial_p(\sigma)$ of a p-simplex $\sigma \in K$ is the alternating sum of its oriented co-dimension 1 faces, and the p-boundary of a p-chain is defined linearly in terms of its constitutive simplices. A p-chain with zero boundary is called a p-cycle, and together they form $Z_p(K) = \operatorname{Ker} \partial_p$. Similarly, the collection of p-boundaries forms $B_p(K) = \operatorname{Im} \partial_{p+1}$. Since $\partial_p \circ \partial_{p+1} = 0$ for all $p \geq 0$, then the quotient

space $H_p(K) = Z_p(K)/B_p(K)$ is well-defined, and called the p-th homology of K with coefficients in $\mathbb{F}$. If $f : K \to [m]$ is a filtration, then the inclusion maps $K_i \subseteq K_{i+1}$ induce linear transformations at the level of homology:

$$H_p(K_1) \to H_p(K_2) \to \cdots \to H_p(K_m) \tag{4}$$

Simplices whose inclusion in the filtration creates a new homology class are called *creators*, and simplices that destroy homology classes are called *destroyers*. The filtration indices of these creators/destroyers are referred to as *birth* and *death* times, respectively. The collection of birth/death pairs (i, j) is denoted $\mathrm{dgm}_p(K, f)$, and referred to as the p-th *persistence diagram* of (K, f). If a homology class is born at K_i and dies entering K_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.

Let $\mathbb{X}$ be a triangulable topological space; that is, so that there exists an abstract simplicial complex K whose geometric realization is homeomorphic to $\mathbb{X}$. Let $f : \mathbb{X} \to \mathbb{R}$ be continuous and 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. if for some p the inclusion-induced homomorphism $H_p(\mathbb{X}_{a-\epsilon}) \to H_p(\mathbb{X}_{a+\epsilon})$ is not an isomorphism for any small enough $\epsilon > 0$. If there are only finitely many of these homological critical values, then f is said to be *tame*. The concept of homological critical points and tameness will be revisited in section 2.2.

2.1 The Reduction Algorithm

In this section we briefly recount the original reduction algorithm introduced in [8], also sometimes called the *standard* algorithm or more explicitly `pHcol` [15]. The pseudocode is outlined in Algorithm 4 in the appendix. Without optimizations, like clearing or implicit matrix reduction, the standard algorithm is very inefficient. Nonetheless, it serves as the foundation of most persistent homology implementations, and its invariants are necessary before introducing both *vineyards* in section 2.2 and our move schedules in section 3.

Given a filtration (K, f) with m simplices, the output of the reduction algorithm is a matrix decomposition $R = DV$, where the persistence diagrams are encoded in R and the generating cycles in the columns of V. To begin the reduction, one first assembles the elementary boundary chains $\partial(\sigma)$ as columns ordered according to f into a $m \times m$ *filtration boundary matrix* D. Setting $V = I$ and $R = D$, one proceeds by performing elementary left-to-right column operations on V and R until the following invariants are satisfied:

Decomposition Invariants:

I1. $R = DV$ where D is the boundary matrix of the filtration (K, f)

I2. V is full-rank upper-triangular

l3. R is *reduced*: if $\mathrm{col}_i(R) \neq 0$ and $\mathrm{col}_j(R) \neq 0$, then $\mathrm{low}_R(i) \neq \mathrm{low}_R(j)$

where $\mathrm{low}_R(i)$ denotes the largest row index of a non-zero entry in column i of R. We call the decomposition satisfying these three invariants *valid*. The persistence diagrams of the corresponding filtration can be determined from the lowest entries in R, once it has been reduced. Note that though the matrices R and V are not unique, the collection of persistent pairings are [8].

It is at times more succinct to restrict to specific sub-matrices of D based on the homology dimension p, and so we write D_p to represent the $d_{p-1} \times d_p$ matrix representing ∂_p (the same notation is extended to R and V). We illustrate the reduction algorithm with an example below.

Example 2.1: Consider a triangle with vertices u, v, w, edges $a = (u, w)$, $b = (v, w)$, $c = (u, v)$, and whose filtration order is given as (u, v, w, a, b, c). Using $\mathbb{Z}_2$ coefficients, the reduction proceeds to compute (R_1, V_1) as follows:

$$
\begin{array}{c} D_1 \\ \\ u \\ v \\ w \end{array}
\begin{array}{c} \begin{array}{cc} a & b \end{array} \\ \begin{bmatrix} 1 & 1 \\ & 1 \\ 1 & \underline{1} \end{bmatrix} \end{array}
\begin{array}{c} I_1 \\ \\ a \\ b \\ c \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \end{array}
\rightarrow
\begin{array}{c} \\ \\ u \\ v \\ w \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ & 1 & \underline{1} \\ 1 & & \end{bmatrix} \end{array}
\begin{array}{c} \\ \\ a \\ b \\ c \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} \end{array}
\rightarrow
\begin{array}{c} \\ \\ u \\ v \\ w \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 1 & \\ & 1 & \\ 1 & & \end{bmatrix} \end{array}
\begin{array}{c} V_1 \\ \\ b \\ a \\ c \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
$$

Since column c in R_1 is 0, the 1-chain indicated by the column c in V_1 represents a dimension 1 cycle. Similarly, the columns at u, v, w in R_0 (not shown) are all zero, indicating three 0-dimensional homology classes are born, two of which are killed by the pivot entries in columns a and b in R_1.

Inspection of the reduction algorithm from [19] suggests that a loose upper bound for the reduction is $O(m^3)$, where m is the number of simplices of the filtration—this bound is in fact tight [4]. Despite this high algorithmic complexity, many variations and optimizations to Algorithm 4 have been proposed over the past decade, see [14, 16, 17] for an overview.

2.2 Vineyards

Consider a homotopy $F(x, t) : \mathbb{X} \times [0, 1] \rightarrow \mathbb{R}$ on a triangulable topological space $\mathbb{X}$, and denote its "snapshot" at a given time-point t by $f_t(x) = F(x, t)$. The snapshot f_0 denotes the initial function at time $t = 0$ and f_1 denotes the function at the last time step. As t varies in $[0, 1]$, the points in $\mathrm{dgm}_p(f_t)$ trace curves in $\mathbb{R}^3$ which, by the stability of persistence, will be continuous if F is continuous and the f_t's are tame. Cohen-Steiner et al. [1] referred to these curves as *vines*, a collection of which forms as *vineyard*—the geometric analogy is meant to act as a guidepost for practitioners seeking to understand the evolution of topological structure over time.

The original purpose of *vineyards*, as described in [2], was to compute a continuous 1-parameter family of persistence diagrams over a time-varying filtration, detecting homological critical events along the way. As homological critical events only occur when the filtration order changes, detecting all such events may be reduced to computing valid decompositions at time points interleaving all changes in the filtration order. For simplexwise filtrations, these changes manifest as transpositions of adjacent simplices, and thus any fixed set of rules that maintains a valid $R = DV$ decomposition under adjacent column transpositions is sufficient to compute persistence dynamically.

To ensure a decomposition is valid, these rules prescribe certain column and row operations to apply to a given matrix decomposition either before, during, or after each transposition. Formally, let S_i^j represent the upper-triangular matrix such that AS_i^j results in adding column i of A to column j of A, and let $S_i^j A$ be the same operation on rows i and j. Similarly, let P denote the matrix so that AP^T permutes the columns of A and PA permutes the rows. Since the columns of P are orthonormal, $P^{-1} = P^T$, then PAP^T performs the same permutation 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.

Example 2.2 To illustrate the basic principles of *vineyards*, we re-use the running example introduced in the previous section. Below, we illustrate the case of exchanging simplices a and b in the filtration order, and restoring RV to a valid decomposition.

$$
\begin{array}{c}
\begin{array}{c} R_1 \\ u \\ v \\ w \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 1 & \\ & 1 & \\ 1 & & \end{bmatrix} \end{array}
\xrightarrow{S_1^2}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & & \\ & 1 & \\ 1 & 1 & \end{bmatrix} \end{array}
\xrightarrow{P}
\begin{array}{c} \begin{array}{ccc} b & a & c \end{array} \\ \begin{bmatrix} 1 & & \\ 1 & & \\ 1 & \underline{1} & \end{bmatrix} \end{array}
\xrightarrow{S_1^2}
\begin{array}{c} \begin{array}{ccc} b & a & c \end{array} \\ \begin{bmatrix} & 1 & \\ 1 & 1 & \\ 1 & & \end{bmatrix} \end{array}
\begin{array}{c} u \\ v \\ w \end{array}
\\[2ex]
\begin{array}{c} V_1 \\ a \\ b \\ c \end{array}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & \underline{1} & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\xrightarrow{S_1^2}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\xrightarrow{P}
\begin{array}{c} \begin{array}{ccc} b & a & c \end{array} \\ \begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\xrightarrow{S_1^2}
\begin{array}{c} \begin{array}{ccc} b & a & c \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\begin{array}{c} b \\ a \\ c \end{array}
\end{array}
$$

Starting with a valid reduction $R = DV$ and 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 S_1^2 on the right to R_1 and V_1. After this operation,

the permutation may be safely applied to V_1. Both before and after the permutation P, 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. Formally, since column operations are the largest complexity operations needed and each column can have potentially $O(m)$ entries, the complexity of *vineyards* is $O(m)$ per transposition. Inspection of the individual cases of the algorithm from [2] shows that any single transposition requires at most two such operations on both R and V. However, several factors can affect the runtime efficiency of the vineyards algorithm. On the positive side, as both the V and R matrices are often sparse, the cost of a given column operation is proportional to the number of non-zero entries in the two columns being modified. Moreover, as a rule of thumb, its been observed that most transpositions require no column operations [19]. On the negative side, one needs to frequently query the non-zero status of various entries in R and V (consider evaluating e.g. Case 1.1 in [2]), which accrues a non-trivial runtime cost due to the quadratic frequency with which they are required.

2.3 Moves

Originally developed to accelerate tracking generators with temporal coherence, Busaryev et al. [12] introduced an extension of the vineyards algorithm which maintains a $R = DV$ decomposition under *move operations*. A move operation $\text{Move}(i, j)$ is a set of rules for maintaining a valid decomposition under the permutation P that moves a simplex σ_i at position i to position j. If $j = i \pm 1$, this operation is an adjacent transposition, and in this sense *moves* generalizes *vineyards*. However, the move framework presented by Busaryev is actually distinct in that it exhibits several attractive qualities not inherited by the *vineyards* approach that warrants further study.

For completeness, we recapitulate the motivation of the moves algorithm from [12]. Let $f : K \to [m]$ denote a filtration of size $m = |K|$ and $R = DV$ its decomposition. Consider the permutation P that moves a simplex σ_i in K to position j, shifting all intermediate simplices $\sigma_{i+1}, \ldots, \sigma_j$ down by one $(i < j)$. To perform this shift, all entries $V_{ik} \neq 0$ with column positions $k \in [i+1, j]$ need to set to zero, otherwise PV is not upper-triangular. We may zero these entries in V using column operations $V(*)$, ensuring invariant I2 (2.1) is maintained, however these operations may render $R' = PR(*)P^T$ unreduced, breaking invariant I3. Of course, we could then reduce R' with additional column operations, but the number of such operations scales $O(|i-j|^2)$, which no more efficient than simply performing the permutation and applying the reduction algorithm to columns $[i, j]$ in R.

To bypass this difficulty, Busaryev et al. observed that since R is reduced, if it contains s pivot entries in the columns $[i, j]$ of R, then R' must also have s pivots. Thus, if column operations render some pivot-column r_k of R

unreduced, then its pivot entry $\text{low}_R(k)$ becomes *free*[4]—if r_k is copied prior to its modification, we may re-use or *donate* its pivot entry to a later column $r_{k+1}, \ldots, r_j$. Repeating this process at most $j - i - 1$ times ensures R' stays reduced in all except possibly at its i-th column. Moreover, since the k-th such operation simultaneously sets $v_{ik} = 0$, V' retains its upper-triangularity.

Example 2.3: We re-use the running example from sections 2.1 and 2.2 to illustrate *moves*. The donor columns of R and V are denoted as d_R and d_V, respectively. Consider moving edge a to the position of edge c in the filtration.

$$\begin{array}{c} d_R \\ u \\ v \\ w \end{array}\!\!\begin{bmatrix} a \\ 1 \\ \\ 1 \end{bmatrix} \begin{array}{c} R \\ u \\ v \\ w \end{array}\!\!\begin{bmatrix} a & b & c \\ 1 & 1 & \\ & & 1 \\ 1 & & \end{bmatrix} \to \begin{array}{c} b \\ \\ \end{array}\!\!\begin{bmatrix} a \\ 1 \\ 1 \end{bmatrix}\!\!\begin{bmatrix} b & c \\ 1 & \\ & 1 \\ 1 & \underline{1} \end{bmatrix} \to \begin{array}{c} c \\ \\ \end{array}\!\!\begin{bmatrix} a \\ \\ \end{bmatrix}\!\!\begin{bmatrix} b & c \\ 1 & 1 \\ 1 & \\ 1 & \underline{1} \end{bmatrix} \xrightarrow{P} \begin{array}{c} a \\ \\ \end{array}\!\!\begin{bmatrix} b & c \\ \\ \end{bmatrix}\!\!\begin{bmatrix} 1 & 1 \\ 1 & \\ 1 & \underline{1} \end{bmatrix} \xrightarrow{d_R} \begin{array}{c} b \\ \\ \end{array}\!\!\begin{bmatrix} c \\ 1 \\ 1 \end{bmatrix}\!\!\begin{bmatrix} a \\ 1 \\ \end{bmatrix}$$

$$\begin{array}{c} d_V \\ a \\ b \\ c \end{array}\!\!\begin{bmatrix} a \\ 1 \\ \\ \end{bmatrix} \begin{array}{c} V \\ a \\ b \\ c \end{array}\!\!\begin{bmatrix} a & b & c \\ 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \to \begin{array}{c} b \\ \\ \end{array}\!\!\begin{bmatrix} a \\ 1 \\ 1 \end{bmatrix}\!\!\begin{bmatrix} b & c \\ 1 & 1 \\ & 1 \\ & \end{bmatrix} \to \begin{array}{c} c \\ \\ \end{array}\!\!\begin{bmatrix} a \\ 1 \\ 1 \end{bmatrix}\!\!\begin{bmatrix} b & c \\ 1 & \\ & 1 \end{bmatrix} \xrightarrow{P} \begin{array}{c} a \\ \\ \end{array}\!\!\begin{bmatrix} b & c \\ \\ \end{bmatrix}\!\!\begin{bmatrix} 1 & \\ & 1 \end{bmatrix} \xrightarrow{d_V} \begin{array}{c} b \\ \\ \end{array}\!\!\begin{bmatrix} c \\ 1 \\ \end{bmatrix}\!\!\begin{bmatrix} a \\ 1 \\ 1 \end{bmatrix}$$

Note that the equivalent permutation using *vineyards* requires 4 column operations on both R_1 and V_1, respectively, whereas a single move operation accomplishes using only 2 column operations per matrix. The pseudo-code for *MoveRight* is given in Algorithm 1 and for *MoveLeft* in Algorithm 2.

Regarding the complexity of move operations, which clearly depend on the sparsity of R and V, we recall the proposition shown in [12]:

Proposition 1 (Busaryev et al. [12])**.** *Given a filtration with n simplices of dimensions $p-1$, p, and $p+1$, let $R = DV$ denote its associated decomposition. Then, the operation* MoveRight(i, j) *constructs a valid decomposition $R' = D'V'$ in $O((|\mathbb{I}| + |\mathbb{J}|)n)$ time, where $\mathbb{I}, \mathbb{J}$ are given by:*

$$|\mathbb{I}| = \sum_{l=i+1}^{j} \mathbb{1}\left(v_l(i) \neq 0\right), \quad \mathbb{J} = \sum_{l=1}^{m} \mathbb{1}\left(\text{low}_R(l) \in [i, j] \text{ and } r_l(i) \neq 0\right)$$

Moreover, the quantity $|\mathbb{I}| + |\mathbb{J}|$ satisfies $|\mathbb{I}| + |\mathbb{J}| \leq 2(j - i)$.

Though similar to *vineyards*, move operations confer additional advantages:

M1: Querying the non-zero status of entries in R or V occurs once per move.

M2: $R = DV$ is <u>not</u> guaranteed to be valid during the movement of σ_i to σ_j.

M3: At most $O(m)$ moves are needed to reindex $f \mapsto f'$

[4]The process of donating pivot columns using $O(1)$ auxiliary storage is similar in spirit to the in-place sorting algorithm *cycle sort*, which is often used to sort permutations in $O(n)$ time.

Algorithm 1 Move Right Algorithm

function RestoreRight(R, V, $\mathbb{I} = \{I_1, I_2, \ldots, I_s\}$, donors = true)
  $(\, d_{low},\, d_R,\, d_V \,) \leftarrow (\, \mathrm{low}_R(I_1),\, \mathrm{col}_R(I_1),\, \mathrm{col}_V(I_1) \,)$
  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 $(\, R,\, V,\, d_R,\, d_V \,)$ **if** donors = true **else** $(\, R,\, V \,)$

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, $\mathbb{J}$, false)
  $(\, R,\, V \,) \leftarrow (\, PRP^T,\, PVP^T \,)$
  $(\, \mathrm{col}_R(j),\, \mathrm{col}_V(j) \,) \leftarrow (\, Pd_R,\, Pd_V \,)$
  return $(\, R,\, V \,)$

Algorithm 2 Move Left Algorithm

function RestoreLeft(R, V, $\mathbb{K} = \{k_1, k_2, \ldots, k_s\}$)
  $(l, r) \leftarrow$ indices $l, r \in \mathbb{K}$ satisfying $l < r$, $\mathrm{low}_R(l) = \mathrm{low}_R(r)$ maximal
  while $\mathrm{low}_R(l) \neq 0$ **and** $\mathrm{low}_R(r) \neq 0$ **do**
    $(\mathrm{col}_R(r), \mathrm{col}_V(r))$ += $(\mathrm{col}_R(l), \mathrm{col}_V(l))$
    $\mathbb{K} \leftarrow \mathbb{K} \setminus l$
    $(l, r) \leftarrow$ indices $l, r \in \mathbb{K}$ satisfying $l < r$, $\mathrm{low}_R(l) = \mathrm{low}_R(r)$ maximal
  return $(\, R,\, V \,)$

function MoveLeft(R, V, i, j)
  $\mathbb{I} \leftarrow \emptyset$
  while $V(k, i) \neq 0$ **for** $k = \mathrm{low}_V(i)$ **where** $j \leq k < i$ **do**
    $(\mathrm{col}_R(i), \mathrm{col}_V(i))$ += $(\mathrm{col}_R(k), \mathrm{col}_V(k))$
    $\mathbb{I} \leftarrow \mathbb{I} \cup k + 1$
  $(R, V) \leftarrow (PRP^T, PVP^T)$
  $\mathbb{J}$ = columns satisfying $\mathrm{low}_R \in [i:j]$ and $\mathrm{row}_R(i) \neq 0$
  $(R, V) \leftarrow$ RestoreLeft($R, V, \mathbb{I}$)
  $(R, V) \leftarrow$ RestoreLeft($R, V, \mathbb{J}$)
  return $(\, R,\, V \,)$

First, consider property M1. Prior to applying any permutation P to the decomposition, it is necessary to remove non-zero entries in V which render P^TVP non-upper triangular, to maintain invariant I2. Using *vineyards*, one must consistently perform $|i-j|-1$ non-zero status queries interleaved between repairing column operations. A move operation groups these status queries into a single pass prior to performing any modifying operations.

Property M2 implies that the decomposition is not fully maintained during the execution of *RestoreRight* and *RestoreLeft* below, which starkly contrasts the *vineyards* algorithm. In this way, we interpret move operations 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*, an equivalent move operation $\text{Move}(i, j)$ generates only one. Indeed, Property M3 directly follows from this fact, as one may simply move each simplex $\sigma \in K$ into its new order $f'(\sigma)$ via insertion sort. Note that the number of valid decompositions produced by *vineyards* is bounded above by $O(m^2)$, as each pair of simplices $\sigma_i, \sigma_j \in K$ may switch its relative ordering at most once during the interpolation from f to f'.

As shown by example 2.3, *moves* can be cheaper than *vineyards* in terms of column operations. However, it is not clear that this is always the case upon inspection of 1, as the usage of a donor column seemingly implies that many $O(m)$ copy operations need to be performed. It turns out that we may handle all such operations except the first in $O(1)$ time, which we formalize below.

Proposition 2. *Let (K, f) denote a filtration of size $|K| = m$ with decomposition $R = DV$ and let T denote the number of column operations needed by the vineyards algorithm to perform the sequence of transpositions:*

$$R = R_1 \xrightarrow{s_1} R_2 \xrightarrow{s_2} \ldots \xrightarrow{s_k} R_{k-1} = R'$$

where s_i denotes the transposition $(i, i+1)$, $i < j$, and $k = |i-j|$. Moreover, let M denote the number of column operations to perform the same update $R \mapsto R'$ with $\text{Move}(i, j)$. Then the inequality $M \leq T$ holds.

Proof First, consider executing the *vineyards* algorithm with a given pair (i, j). As there are at most 2 column operations, any contiguous sequence of transpositions $(i, i+1), (i+1, i+2), \ldots (j-1, j)$ induces at most $2(|i-j|)$ column operations in both R and V, giving a total of $4(|i-j|)$ column operations.

Now consider a single MoveRight(i,j) outlined in Algorithm 1. Here, the dominant cost again are the column operations (line 5). Though we need an extra $O(m)$ storage allocation for the donor columns d_* prior to the movement, notice that assignment to and from d_* (lines (4), and (7) in RestoreRight of MoveRight, respectively) requires just $O(1)$ time via a pointer swapping argument. That is, when $d'_{\text{low}} < d_{\text{low}}$, instead of copying $\text{col}_*(k)$ to d'_*—which takes

$O(m)$ time—we instead swap their column pointers in $O(1)$ prior to column operations. After the movement, d_* contains the newly modified column and $\mathrm{col}_*(k)$ contains the unmodified donor d'_*, so the final donor swap also requires $O(1)$ time. Since at most one $O(m)$ column operation is required for each index in $[i, j]$, moving a column from i to j where $i < j$ requires at most $2(|i-j|)$ column operations for both R and V. The claimed inequality follows. □

As a final remark, we note that the combination of *MoveRight* and *MoveLeft* enable efficient simplex additions or deletions to the underlying complex. In particular, given K and a decomposition $R = DV$, obtaining a valid decomposition $R' = D'V'$ of $K' = K \cup \{\sigma\}$ can be achieved by appending its requisite elementary chains to D and V, reducing them, and then executing *MoveLeft*$(m+1, i)$ with $i = f'(\sigma)$. Dually, deleting a simplex σ_i may be achieved via *MoveRight* by moving i-th to the end of the decomposition and dropping the corresponding columns.

3 Our contribution: Move Schedules

We begin with a brief overview of the pipeline to compute the persistence diagrams of a discrete 1-parameter family $(f_1, f_2, \ldots, f_n)$ of filtrations. Without loss of generality, assume each filtration $f_i : K \to [m]$ is a simplexwise filtration of a fixed simplicial complex K with $|K| = m$, and let (f, f') denote any pair of such filtrations. Our strategy to efficiently obtain a valid RV decomposition of the filtration (K, f') from a given decomposition of (K, f) is to decompose a fixed bijection $\rho : [m] \to [m]$ satisfying $f' = \rho \circ f$ into a *schedule* of updates:

Definition 1 (Schedule)**.** *Given a pair of filtrations $(K, f), (K, f')$ and $R = DV$ the initial decomposition of (K, f), a* schedule $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ *is a sequence of permutations satisfying:*

$$R = D_0V_0 \xrightarrow{s_1} D_1V_1 \xrightarrow{s_2} \ldots \xrightarrow{s_d} D_dV_d = R' \tag{5}$$

where, for each $i \in [d]$, $R_i = D_iV_i$ is a valid decomposition respecting invariants 2.1, and R' is a valid decomposition for (K, f').

To produce this sequence of permutations $\mathcal{S} = (s_1, \ldots, s_d)$ from ρ, our approach is as follows: define $q = (\rho(1), \rho(2), \ldots, \rho(m))$. We compute a longest increasing subsequence $\mathrm{LIS}(q)$, and then use this subsequence to recover a longest common subsequence (LCS) between f and f', which we denote later with $\mathrm{LCS}(f, f')$. We pass q, $\mathrm{LIS}(q)$, and a "greedy" heuristic for picking moves h to our scheduling algorithm (to be defined later), which returns as output an ordered set of move permutations $\mathcal{S}$ of minimum size satisfying (5)—a *schedule* for the pair (f, f'). These sequence of steps is outlined in Algorithm 3 below.

Though using the LCS between a pair of permutations induced by simplexwise filtrations is a relatively intuitive way of producing a schedule of move-type decomposition updates, it is not immediately clear whether such

Algorithm 3 Scheduling algorithm

Require: Filtration pair $\mathcal{F} = \{f_0, f_1\}$ and valid $R = DV$ for (K, f_0)
Ensure: Valid $R = DV$ is computed for (K, f_1)

procedure MoveSchedule($\mathcal{F} = (f_0, f_1)$, R, V)
  Fix $\rho \in S_m$ such that $f_1 = \rho \circ f_0$ ▷ $O(m)$
  $q \leftarrow (\rho(1), \rho(2), \ldots, \rho(m))$ ▷ $O(m)$
  $\text{lis}_q \leftarrow \text{LIS}(q)$ ▷ $O(m \log\log m)$
  $h \leftarrow$ `greedy` ▷ Heuristic for picking moves
  $\mathcal{S} \leftarrow \text{Schedule}(q, \text{lis}_q, h)$ ▷ $O(d^2 \log m)$
  for (i, j) **in** $\mathcal{S}$ **do** ▷ $|\mathcal{S}| = d$
    $(R, V) \leftarrow$ **if** $i < j$ MoveRight(i, j) **else** MoveLeft(i, j)

an algorithm has any computational benefits compared to *vineyards*. Indeed, since *moves* is a generalization of *vineyards*, several important questions arise when considering practical aspects of how to implement Algorithm 3, such as e.g. how to pick a "good" heuristic h, how expensive can the Schedule algorithm be, or how $|\mathcal{S}|$ scales with respect to the size of the complex K. We address these issues in the following sections.

3.1 Continuous setting

In the *vineyards* setting, a given homotopy $F : K \times [0,1] \to \mathbb{R}$ continuously interpolating between (K, f) and (K, f') is discretized into a set of critical events that alter the filtration order. As F determines the number of distinct filtrations encountered during the deformation from f to f', a natural question is whether such an interpolation can be modified so as to minimize $|\mathcal{S}|$—the number of times the decomposition is restored to a valid state. Towards explaining the phenomenon exhibited in Figure 2, we begin by analyzing a particular class of interpolation schemes in order to establish an upper bound on this quantity.

Let $F : K \times [0,1] \to \mathbb{R}$ be a homotopy of x-monotone curves[5] between the filtrations $f, f' : K \to [m]$ whose function $t \mapsto F(\sigma, t)$ is continuous and satisfies $f(\sigma) = F(\sigma, 0)$ and $f'(\sigma) = F(\sigma, 1)$ for every $\sigma \in K$. Note this family includes the straight-line homotopy $F(\sigma, t) = (1-t)f(\sigma) + tf'(\sigma)$, studied in the original *vineyards* paper [2]. If we assume that each pair of curves $\big(t, F(\cdot, t)\big) \subset [0,1] \times \mathbb{R}$ intersect in at most one point—at which they cross—the continuity and genericity assumptions on F imply that for $\sigma, \mu \in K$ distinct, the curves $t \mapsto F(\sigma, t)$ and $t \mapsto F(\mu, t)$ intersect if and only if $f(\sigma) > f(\mu)$ and $f'(\sigma) < f'(\mu)$, or $f(\sigma) < f(\mu)$ and $f'(\sigma) > f'(\mu)$. In other words, the number of crossings in F is exactly the *Kendall-τ* distance [27] between f and f':

$$K_\tau(f, f') = \frac{1}{2} \left| \left\{ (\sigma, \mu) \mid \mathsf{sign}\big(f(\sigma) - f(\mu)\big) \neq \mathsf{sign}\big(f'(\sigma) - f'(\mu)\big) \right\} \right| \tag{6}$$

[5] This term has been used in reference to parameterized curves whose behavior with respect to a certain restricted set of geometric predicates is invariant, see [26].

After slightly perturbing F if necessary, we can further assume that its crossings occur at $k = K_\tau(f, f')$ distinct time points $0 < t_1 < \cdots < t_k < 1$. Let $t_0 = 0$, $t_{k+1} = 1$ and fix $a_i \in (t_i, t_{i+1})$ for $i = 0, \ldots, k$. Then, the order in K induced by $\sigma \mapsto F(\sigma, a_i)$ defines a filtration $f_i : K \to [m]$ so that $f_0 = f$, $f_k = f'$ and $\mathcal{F} = (f_0, f_1, \ldots, f_k)$ is the ordered sequence of all distinct filtrations in the interpolation from f to f' via F.

The continuity of the curves $t \mapsto F(\cdot, t)$ and the fact that t_i is the sole crossing time in the interval (t_{i-1}, t_{i+1}), imply that the permutation ρ_i transforming f_{i-1} into f_i, i.e. so that $f_i = \rho_i \circ f_{i-1}$, must be (in cycle notation) of the form $\rho_i = (\ell_i \;\; \ell_i + 1)$ for $1 \leq \ell_i < m$. In other words, ρ_i is an adjacent transposition for each $i = 1, \ldots, k$. Observe the size of the ordered sequence of adjacent transpositions $S_F = (\rho_1, \rho_2, \ldots, \rho_k)$ defined from the homotopy F above is exactly $K_\tau(f, f')$. On the positive side, the reduction of schedule planning to crossing detection implies the former can be solved optimally in output-sensitive $O(m \log m + k)$ time by several algorithms [26], where k is the output-sensitive term and m is the number of simplices in the filtration(s). On the negative side, $k = K_\tau(f, f')$ scales in size to $\sim O(m^2)$ in the worst case, achieved when $f' = -f$. As mentioned in 2.2, this quadratic scaling induces a number of issues in the practical implementations of the *vineyards* algorithm

Remark 1. *The grayscale image data example from section 1.2 exhibits this quadratic scaling. Indeed, the Freudenthal triangulation of the* 9×9 *grid contains* $(81, 208, 128)$ *simplices of dimensions* $(0, 1, 2)$*, respectively. Therefore,* $m = 417$ *and* $|S_F| \leq \frac{1}{2}m(m-1) = 86,736$*. As the homotopy given by the video is varied,* $\approx$ 70,000 *transpositions are generated, approaching the worst case upper bound due to the fact that* f' *is nearly the reverse of* f*.*

If our goal is to decrease $|S_F|$, one option is to *coarsen* S_F to a new schedule $\widetilde{S}_F$ by e.g. collapsing contiguous sequences of adjacent transpositions to moves, via the map:

$$(i, i+1)(i+1, i+2)\cdots(j-1, j) \mapsto (j, i+1, \cdots, j-1, i) \quad \text{if } i < j \tag{7}$$

Clearly $|\widetilde{S}_F| \leq |S_F|$ and the associated coarsened $\widetilde{S}_F$ requires just $O(m)$ time to compute. However, the coarsening depends entirely on the initial choice of F and the quadratic upper bound remains—it is always possible that there are no contiguous subsequences to collapse. Suggesting one must either abandon the continuous setting or make stronger assumptions on F to have any hope of keeping $|S_F| \sim O(m)$ in size.

3.2 Discrete setting

Contrasting the continuous-time setting, if we discard the use of a homotopy interpolation and allow move operations in any order, we obtain a trivial upper bound of $O(m)$ on the schedule size: simply move each simplex in K from its position in the filtration given by f to the position given by f'—which we call

the *naive strategy*. In particular, in losing the interpolation interpretation it is no longer clear the $O(m)$ bound is tight. Indeed, the "intermediate" filtrations need no longer even respect the face poset of the underlying complex K. In this section, we investigate these issues from a combinatorial perspective.

Let S_m denote the symmetric group. Given two fixed permutations $p, q \in S_m$ and a set allowable permutations $\Sigma \subseteq S_m$, a common problem is to find a sequence of permutations $s_1, s_2, \ldots, s_d \in \Sigma$ whose composition satisfies:

$$s_d \circ \cdots \circ s_2 \circ s_1 \circ p = q \tag{8}$$

Common variations of this problem include finding such a sequence of minimal length (d) and bounding the length d as a function of m. In the latter case, the largest lower bound on d is referred to as the *distance* between p and q with respect to Σ. A sequence $S = (s_1, s_2, \ldots, s_d)$ of operations $s \in \Sigma \subseteq S_m$ mapping $p \mapsto q$ is sometimes called a *sorting of* p. When p, q are interpreted as strings, these operations $s \in \Sigma$ are called *edit operations*. The minimal number of edit operations $d_\Sigma(p, q)$ needed to sort $p \mapsto q$ with respect to Σ is referred to as the *edit distance* [28] between p and q. We denote the space of sequences transforming $p \mapsto q$ using d permutations in $\Sigma \subseteq S_m$ with $\Phi_\Sigma(p, q, d)$. Note the choice of Σ defines the type of distance being measured—otherwise if $\Sigma = S_m$, then $d_\Sigma(p, q) = 1$ trivially for any $p \neq q \in S_m$.

Perhaps surprisingly, small changes to set of allowable edit operations Σ dramatically affect both the size of $d_\Sigma(p, q)$ and the difficulty of obtaining a minimal sorting. 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 [29]. 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) yields a sorting of size $K_\tau(p, q)$.

Here we consider sorting by moves. Using permutations, a *move operation* m_{ij} that moves i to j in $[m]$, for $i < j$, corresponds to the circular rotation:

$$m_{ij} = \begin{pmatrix} 1 \cdots i-1 & \overline{i \quad i+1 \quad \cdots \quad j-1 \; j} & j+1 \cdots m \\ 1 \cdots i-1 & \overline{i+1 \quad \cdots \quad j-1 \quad j \quad i} & j+1 \cdots m \end{pmatrix} \tag{9}$$

In cycle notation, this corresponds to the cyclic permutation:

$$m_{ij} = (\; i \;\; j \;\; j\text{-}1 \;\; \ldots \;\; i\text{+}2 \;\; i\text{+}1 \;) \tag{10}$$

Observe that a move operation can be interpreted as a paired delete-and-insert operation, i.e. $m_{ij} = (\mathrm{ins}_j \circ \mathrm{del}_i)$, where del_i denotes the operation that deletes the character at position i and ins_j the operation that inserts the same character at position j. Thus, sorting by move operations can be interpreted as finding a minimal sequence of edits where the only operations

allowed are (paired) insertions and deletions—this is exactly the well known *Longest Common Subsequence* (LCS) distance. Between strings p, q of sizes m and n, the LCS distance is given by [28]:

$$\mathrm{d}_{\mathrm{lcs}}(p, q) = m + n - 2|\mathrm{LCS}(p, q)| \tag{11}$$

With this insight in mind, we obtain the following bound on the minimum size of a sorting (i.e. schedule) using moves and the complexity of computing it.

Proposition 3 (Schedule Size)**.** *Let $(K, f), (K, f')$ denote two filtrations of size $|K| = m$. Then, the smallest move schedule S^* reindexing $f \mapsto f'$ has size:*

$$|S^*| = d = m - |\mathrm{LCS}(f, f')|$$

where we use $\mathrm{LCS}(f, f')$ to denote the LCS of the permutations of K induced by f and f'.

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

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

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

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

where $\iota = [m]$, the identity permutation. Suppose we are given two permutations $p, q \in S_n$ and we seek to compute $\mathrm{LCS}(p, q)$. Consider the permutation $p' = q^{-1} \circ p$. Since the LCS distance is a valid edit distance, if $|\mathrm{LCS}(p, q)| = k$, then $|\mathrm{LCS}(p', \iota)| = k$ as well. Notice that ι is strictly increasing and that any common subsequence p' has with ι must also be strictly increasing. The optimality of d follows from the optimality of the well-studied LCS problem [30]. □

For any pair of general string inputs of sizes n and m, respectively, the LCS between them is computable in $O(mn)$ with dynamic programming, and there is substantial evidence that the complexity cannot be much lower than this [31]. In our setting, however, observe the total orders of any pair of simplexwise filtrations $(K, f), (K, f')$ of the same underlying complex K may be thought

of a permutations in S_m, in which case d_{LCS} reduces further to the *permutation edit distance* problem. This special type of edit distance has additional structure to it, which we demonstrate below.

Corollary 1. *Let $(K, f), (K, f')$ denote two filtrations of size $|K| = m$, and let S^* denote schedule of minimal size reindexing $f \mapsto f'$. Then $|S^*| = d$ can be determined in $O(m \log\log m)$ time.*

Proof By the same reduction from Proposition 3, the problem of computing $\mathrm{LCS}(f, f')$ reduces to the problem of computing the *longest increasing subsequence* (LIS) of a particular permutation $p' \in S_m$, which can done in $O(m \log\log m)$ time using van Emde Boas trees [32] □

Reduced complexity is not the only immediate benefit from Proposition 3; by the same reduction to the LIS problem, we obtain the worst-case bounds on $\mathcal{S}$ in expectation.

Corollary 2. *If $(K, f), (K, f')$ are random filtrations of a common complex K of size m, then the expected size of longest common subsequence $\mathrm{LCS}(f, f')$ between f, f' is no larger than $m - \sqrt{m}$, 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:

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

A large body of work dates back at least 50 years has focused on estimating this quantity, which is sometimes called the *Ulam-Hammersley* problem. Seminal work by Baik et al. [33] established that as $m \to \infty$:

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

where $c = -1.77108....$ Moreover, letting $m \to \infty$, 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 $m \to \infty$. Using the reduction from above to show that $\mathrm{LCS}(p, q) \Leftrightarrow \mathrm{LIS}(p')$, the claimed bound follows. □

Remark 2. *Note the quantity from Corollary 2 captures the size of S^* between pairs of uniformly sampled permutations, as opposed to uniformly sampled filtrations, which have more structure due to the face poset. However, Boissonnat [34] prove the number of distinct filtrations built from a k-dimensional simplicial complex K with m simplices and t distinct filtration values is* at least $\lfloor \frac{t+1}{k+1} \rfloor^m$. *Since this bound grows similarly to $m!$ when $t \sim O(m)$ and $k << m$ fixed, $d \approx n - \sqrt{n}$ is not too pessimistic a bound between random filtrations.*

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, shrinking d substantially. For example, for the complex from Section 1.2 with $m = 417$ simplices, the average size of the LCS across the 10 evenly spaced filtrations was 343, implying $d \approx 70$ permutations needed on average to update the decomposition between adjacent time points.

We conclude this section with the main theorem of this effort: an output-sensitive bound on the computation of persistence dynamically.

Theorem 1. *Given a pair of filtrations $(K, f), (K, f')$, a decomposition $R = DV$ of K, and a sequence $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ of cyclic 'move' permutations $s_k = (i_k, j_k)$ satisfying $i_k < j_k$ for all $k \in [d]$, computing the updates:*

$$R = D_0 V_0 \xrightarrow{s_1} D_1 V_1 \xrightarrow{s_2} \ldots \xrightarrow{s_d} D_d V_d = R' \tag{12}$$

where $R' = D_d V_d$ denotes a valid decomposition of (K, f') requires $O(\nu)$ column operations, where ν depends on the sparsity of the intermediate entries $V_1, V_2, \ldots, V_d$ and $R_1, R_2, \ldots, R_d$:

$$\nu = \sum_{k=1}^{d} (|\mathbb{I}_k| + |\mathbb{J}_k|), \qquad \textit{where } |\mathbb{I}_k|, |\mathbb{J}_k| \textit{are given in Proposition 1}$$

Moreover, the size of a minimal such $\mathcal{S}$ can be determined in $O(m \log\log m)$ time and $O(m)$ space.

Proof Proposition 3 yields the necessary conditions for constructing $\mathcal{S}$ with optimal size d in $O(m \log\log m)$ time and $O(m)$. The definition of ν follows directly from Algorithm 1. □

3.3 Constructing schedules

While it is clear from the proof in Proposition 3 that one may compute the LCS between two permutations $p, q \in S_m$ in $O(m \log\log m)$ time, it is not immediately clear how to obtain a sorting $p \mapsto q$ from a given $\mathcal{L} = \mathrm{LCS}(p, q)$ in an efficient way. We outline below a simple procedure which constructs such a

sorting $\mathcal{S} = (s_1, \ldots, s_d)$ in $O(dm \log m)$ time and $O(m)$ space, or $O(m \log m)$ time and $O(m)$ space per update in the online setting.

First, we require a few definitions. Recall that a *sorting* $\mathcal{S}$ with respect to two permutations $p, q \in S_m$ is an ordered sequence of permutations $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ satisfying $q = s_d \circ \ldots s_1 \circ p$. By definition, a subsequence in $\mathcal{L}$ common to both p and q satisfies:

$$p^{-1}(\sigma) < p^{-1}(\tau) \implies q^{-1}(\sigma) < q^{-1}(\tau) \quad \forall \sigma, \tau \in \mathcal{L} \tag{13}$$

where $p^{-1}(\sigma)$ (resp. $q^{-1}(\sigma)$) denotes the position of σ in p (resp. q). Thus, obtaining a sorting $p \mapsto q$ of size $d = m - |\mathcal{L}|$ reduces to applying a sequence of moves in the complement of $\mathcal{L}$. Formally, we define a permutation $s \in S_m$ as a *valid* operation with respect to a fixed pair $p, q \in S_m$ if:

$$|\mathrm{LCS}(s \circ p, q)| = |\mathrm{LCS}(p, q)| + 1 \tag{14}$$

The problem of constructing a sorting $\mathcal{S}$ of size d thus reduces to the problem of choosing a sequence of d valid moves, which we call a *valid sorting*. To do this efficiently, let $\mathcal{U}$ denote an ordered set-like data structure that supports the following operations on elements $\sigma \in M$ from the set $M = \{0, 1, \ldots, m+1\}$:

1 $\mathcal{U} \cup \sigma$—inserts σ into $\mathcal{U}$,
2 $\mathcal{U} \setminus \sigma$—removes σ from $\mathcal{U}$,
3 $\mathcal{U}_{\mathrm{succ}}(\sigma)$—obtain the successor of σ in $\mathcal{U}$, if it exists, otherwise return $m+1$
4 $\mathcal{U}_{\mathrm{pred}}(\sigma)$—obtain the predecessor of σ in $\mathcal{U}$, if it exists, otherwise return 0

Given $\mathcal{U}$, a valid sorting can be constructed by querying and maintaining information about the LCS in $\mathcal{U}$. To see this, suppose $\mathcal{U}$ contains the current LCS between two permutations p and q. By definition of the LCS, we have:

$$p^{-1}(\mathcal{U}_{\mathrm{pred}}(\sigma)) < p^{-1}(\sigma) < p^{-1}(\mathcal{U}_{\mathrm{succ}}(\sigma)) \tag{15}$$

for every $\sigma \in \mathcal{U}$. Now, suppose we choose some element $\sigma \notin \mathcal{U}$ which we would like to add to the LCS. If $p^{-1}(\sigma) < p^{-1}(\mathcal{U}_{\mathrm{pred}}(\sigma))$, then we must move σ to the right of its predecessor in p such that (15) holds. Similarly, if $p^{-1}(\mathcal{U}_{\mathrm{succ}}(\sigma)) < p^{-1}(\sigma)$, then we must move σ left of its successor in p. Assuming the structure $\mathcal{U}$ supports all of the above operations in $O(\log m)$ time, we easily deduce a $O(dm \log m)$ algorithm for obtaining a valid sorting.

3.4 Minimizing schedule cost

The algorithm outlined in section 3.3 is a sufficient for generating move schedules of minimal cardinality: any schedule of moves S sorting $f \mapsto f'$ above is guaranteed to have size $|S| = m - |\mathrm{LCS}(f, f')|$, and the reduction to the permutation edit distance problem ensures this size is optimal. However, as with the *vineyards* algorithm, certain pairs of simplices cost more to exchange depending on whether they are critical pairs in the sense described in [2], resulting in a large variability in the cost of randomly generated schedules. This variability

is undesirable in practice: we would like to generate a schedule which not only small in size, but is also efficient in terms of its required column operations.

3.4.1 Greedy approach

Ideally, we would like to minimize the cost of a schedule $\mathcal{S} \in \Phi_\Sigma(p, q, d)$ directly, which recall is given by the number of non-zeros at certain entries in R and V:

$$\text{cost}(\mathcal{S}) = \sum_{k=1}^{d} |\mathbb{I}_k| + |\mathbb{J}_k| \tag{16}$$

where $|\mathbb{I}| + |\mathbb{J}|$ are the quantities from Proposition 1. Globally minimizing the objective (16) directly is difficult due to the changing sparsity of the intermediate matrices R_k, V_k. One advantage of the *moves* framework is that the cost of a single move on a given $R = DV$ decomposition can be determined efficiently prior to any column operations. Thus, it is natural to consider whether one could minimize (16) by greedily choosing the lowest cost move each step. Unfortunately, not only does this approach does not yield a minimal cost solution, we give a counter-example in the appendix (A.2) demonstrating such a greedy procedure may lead to arbitrarily bad behavior.

3.4.2 Proxy objective

In light of section 3.4.1, we seek an alternative objective that correlates with (16) and does not depend on the entries in the decomposition. Given a pair of filtrations $(K, f), (K, f')$, a natural schedule $\mathcal{S} \in \Phi(f, f', d)$ of cyclic permutations $(i_1, j_1), (i_2, j_2), \ldots, (i_d, j_d)$ is one minimizing the upper bound:

$$\widetilde{\text{cost}}(\mathcal{S}) = \sum_{k=1}^{d} 2|i_k - j_k| \geq \sum_{k=1}^{d} (|\mathbb{I}_k| + |\mathbb{J}_k|) \tag{17}$$

Unfortunately, even obtaining an optimal schedule $\mathcal{S}^* \in \Phi(f, f', d)$ minimizing (17) does not appear tractable due to its similarity with the k-layer crossing minimization problem, which is NP-hard for k sets of permutations when $k \geq 4$ [35]. For additional discussion on the relationship between these two problems, see section A.3 in the appendix.

In light of the discussion above, we devise a *proxy* objective function based on the Spearman distance [27] which we observed is both efficient to optimize and effective in practice. The *Spearman footrule distance* $F(p, q)$ between two $p, q \in S_m$ is an ℓ_1-type distance for measuring permutation disarrangement; it is classically defined as:

$$F(p, q) = \sum_{i=1}^{m} |\, p(i) - q(i) \,| = \sum_{i=1}^{m} |\, i - (q^{-1} \circ p)(i) \,| \tag{18}$$

Like K_τ, F forms a metric on S_m and is invariant under relabeling. Our motivation for considering the footrule distance is motivated by the fact that F recovers $\widetilde{\text{cost}}(\mathcal{S})$ when (p, q) differ by a cyclic permutation (i.e. $F(p, m_{ij} \circ p) = 2|i - j|$) and by its usage on similar[6] combinatorial optimization problems. To adapt F to sortings, we decompose F additively via the bound:

$$\hat{F}_{\mathcal{S}}(p, \iota) = \sum_{i=1}^{d-1} F(\hat{s}_i \circ p, \hat{s}_{i+1} \circ p) \geq F(p, \iota) \tag{19}$$

where $\hat{s}_i = s_i \circ \cdots \circ s_2 \circ s_1$ denote the composition of the first i permutations of a sorting $S = (s_1, \ldots, s_d)$ that maps $p \mapsto \iota$. As a heuristic to minimize (19), we greedily select the optimal $k \in \mathcal{L}$ at each step which minimizes the Spearman distance to the identity permutation:

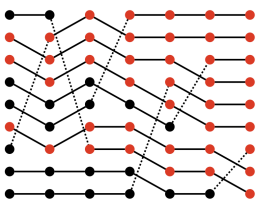
$$k_{\text{greedy}} = \arg\min_{k \in \mathcal{D}} F(m_{ij} \circ p, \iota), \quad i = p^{-1}(k) \tag{20}$$

Note that equality between F and $\hat{F}_{\mathcal{S}}$ (19) is achieved when the displacement of each $\sigma \in p$ between its initial position in p to its value is non-increasing with every application of s_i, which in general not guaranteed using the schedule construction method in section 3. To build intuition for how this heuristic interacts with Algorithm 5, we show a purely combinatorial example below.

Example: The permutation $p \in S_m$ to sort to the identity $p \mapsto \iota$ is given as a sequence $(4, 2, 7, 1, 8, 6, 3, 5, 0)$ and a precomputed LIS $\mathcal{L} = (1, 3, 5)$, which is highlighted in red. On the left, a table records permuted sequences $\{\hat{s}_i \circ p\}_{i=0}^{d}$,

4	2	7	1	8	6	3	5	0
4	2	1	8	6	3	5	7	0
4	2	0	1	8	6	3	5	7
4	2	0	1	6	3	5	7	8
2	0	1	6	3	4	5	7	8
2	0	1	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

30	30	30	24	30	26	28	30	30	26
24	22	22	24	20	20	24	24	24	20
20	18	18	20	20	14	18	20	20	20
14	10	12	14	14	10	14	14	14	14
10	6	10	10	4	10	10	10	10	10
4	0	4	4	4	4	4	4	4	4
0	0	0	0	0	0	0	0	0	0



given on rows $i \in \{0, 1, \ldots, 6\}$, for the moved elements $(7, 0, 8, 4, 6, 2)$ (highlighted in yellow). In the middle, we show the Spearman distance cost (20) associated with both each permuted sequence (left column) and with each candidate permutation $m_{ij} \circ p$ (rows). Note that elements $\sigma \in \mathcal{L}$ (red) induce identity permutations that do not modify the cost, and thus the minimization is restricted to elements $\sigma \in p \setminus \mathcal{L}$ (black). The right plot shows the connection to the bipartite crossing minimization problem discussed in section 3.4.1.

[6] F is often used to approximate K_τ in *rank aggregation* problems due to the fact that computing the [Kemini] optimal rank aggregation is NP-hard with respect to K_τ, but only polynomial time with respect to F [36]. Note F bounds K_τ via the inequalities $K_\tau(p, q) \leq F(p, q) \leq 2K_\tau(p, q)$ [27].

3.4.3 Heuristic Computation

We now introduce an efficient $O(d^2 \log m)$ algorithm for constructing a move schedule that greedily minimizes (19). The algorithm is purely combinatorial, and is motivated by the simplicity of updating the Spearman distance between sequences which differ by a cyclic permutation.

First, consider an array $\mathcal{A}$ of size m which provides $O(1)$ access and modification, initialized with the *signed displacement* of every element in p to its corresponding position in q. In the case where $q = \iota$, note $\mathcal{A}(i) = i - p(i)$, thus $F(p, \iota)$ is given by the sum of the absolute values of the elements in $\mathcal{A}$. In other words, computing $F(m_{ij} \circ p, \iota)$ in $O(\log m)$ time corresponds to updating an arrays *prefix sum* under element insertions and deletions, which is easily solved by many data structures (e.g. segment trees).

Because the values of $\mathcal{A}$ are signed, the reduction to prefix sums is not exact: it is not immediately clear how to modify $|i-j|$ elements in $O(\log m)$ time. To address this, observe that at any point during the execution of Algorithm 5, a cyclic permutation changes each value of $\mathcal{A}$ in at most three different ways:

$$\mathcal{A}(m_{ij} \circ p) = \begin{cases} \mathcal{A}(k) \pm |i-j| & p^{-1}(k) = i \\ \mathcal{A}(k) \pm 1 & i < p^{-1}(k) \leq j \\ \mathcal{A}(k) & \text{otherwise} \end{cases}$$

Thus, $\mathcal{A}$ may be partitioned into at most four contiguous intervals each upon which the update is constant and we need only apply a constant number of *range updates* to $\mathcal{A}$. Such updates are known to require $O(\log m)$ time using e.g. an *implicit treap* data structure [37]. Since single element modifications, deletions, insertions, and constant range updates can all be achieved in $O(\log m)$ expected time with such a data structure, we conclude that equation (20) may be solved in just $O(d^2 \log m)$ time.

Example: We re-use the previous example of sorting the sequence $p = (4, 2, 7, 1, 8, 6, 3, 5, 0)$ to the identity $\iota = [m]$ using a precomputed LIS $\mathcal{L} = (1, 3, 5)$. The left table we show the same sorting as before, with elements chosen to move highlighted in yellow; on the right, a table showing the entries of $\mathcal{A}$ are recorded. The colors of each entry in the right table indicate its

4	2	7	1	8	6	3	5	0
4	2	1	8	6	3	5	7	0
4	2	0	1	8	6	3	5	7
4	2	0	1	6	3	5	7	8
2	0	1	6	3	4	5	7	8
2	0	1	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

30	-4	-1	-5	2	-4	-1	3	2	8
24	-4	-1	1	-5	-2	2	1	0	8
20	-4	-1	2	2	-4	-1	3	2	1
14	-4	-1	2	2	-2	2	1	0	0
10	-2	1	1	-3	1	1	1	0	0
4	-2	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

partition: green values are unchanged between permutations, blue values are modified by ± 1, and orange values are modified arbitrarily. As before, the ith entry of the column on the left-side shows $F(\hat{s}_i \circ p, \iota)$. Small black lines are used to show the movement of the entries in $\mathcal{A}$ which change by ± 1.

4 Applications and Experiments

4.1 Video data

A common application of persistence is characterizing topological structure in image data. Since a set of "snapshot" frames of a video can be equivalently thought of as discrete 1-parameter family, our framework provides a natural extension of the typical image analysis to video data. To demonstrate the benefit of scheduling and the scalability of the greedy heuristic, we perform two performance tests on the video data from section 1.2: one to test the impact of repairing the decomposition less and one to measure the asymptotic behavior of the greedy approach.

In the first test, we fix a grid size of 9×9 and record the cumulative number of column operations needed to compute persistence dynamically across 25 evenly-spaced time points using a variety of scheduling strategies. The three strategies we test are the greedy approach from section 3.4.2, the "simple" approach which uses upwards of $O(m)$ move permutations via selection sort, and a third strategy which interpolates between the two. To perform this interpolation, we use a parameter $\alpha \in [0, 1]$ to choose $m - \alpha \cdot d$ random simplices to move using the same construction method outlined in section 3.3. The results are summarized in the left graph on Figure 3, wherein the mean schedule cost of the random strategies are depicted by solid lines. To capture the variation in performance, we run 10 independent iterations and shade the upper and lower bounds of the schedule costs. As seen in Figure 3, while using less move operations (lower α) tends to reduce column operations, constructing *random* schedules of minimal size is no more competitive than the selection sort strategy. This suggests that efficient schedule construction needs to account for the structure of performing several permutations in sequence, like the greedy heuristic we introduced, to yield an adequate performance boost.

In the second test, we aim to measure the asymptotics of our greedy LCS-based approach. To do this, we generated 8 video data sets again of the expanding annulus outlined in section 1.2, each of increasing grid sizes of 5×5, 6×6, $\ldots$, 12×12. For each data set, we compute persistence over the duration of the video, again testing five evenly spaced settings of $\alpha \in [0, 1]$—the results are shown in the right plot of Figure 3. On the vertical axis, we plot the total number of column operations needed to compute persistence across 25 evenly-spaced time points *as a ratio* of the data set size (m); we also show the regression curves one obtains for each setting of α. As one can see from the Figure, the cost of using the greedy heuristic tends to increase sub-linearly as a function of the data set size, suggesting the move scheduling approach is indeed quite scalable. Moreover, schedules with minimal size tended to be

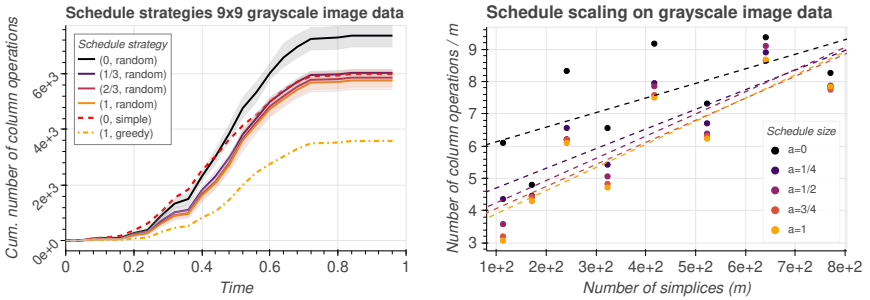


Figure 3: Performance comparison between various scheduling strategies. On the left, the cumulative column operations required to compute the 1-parameter family is shown for varying schedule sizes (d) and strategies. On the right, both the size of the schedule and the data set (m) are varied.

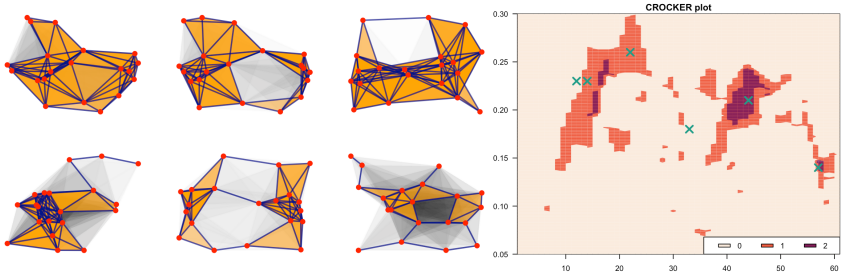


Figure 4: A crocker plot (right) depicts the evolution of dimension $p = 1$ Betti curves over time. The green X marks correspond chronologically to the complexes (left), in row-major order. The large orange and purple areas depict 1-cycles persisting in both space (y-axis) and time (x-axis).

cheaper than otherwise, confirming our initial hypothesis that repairing the decomposition less can lead to substantial reductions at runtime.

4.2 Crocker stacks

There are many challenges to characterizing 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 [24, 25].

Acknowledging this, Topaz et al. [24] proposed the use of a 2-dimensional summary visualization, called a *crocker*[7] plot. In brief, a crocker plot is a contour plot of 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 displays changes to $\beta_p^\bullet(\tau)$ as a function of τ. A example of a crocker plot generated from the simulation described below is given in Figure 4. 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 1.1. 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.

Xian et al. [25] 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 2 (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:*

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

and f_M satisfies $f_M(t,\epsilon,\alpha') \leq f_M(t,\epsilon,\alpha)$ for all $0 \leq \alpha \leq \alpha'$.

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 [25]. Thus, computing crocker stacks reduces to computing the persistence of a (time-varying) family of filtrations.

To illustrate the applicability of our method, we test the efficiency of computing these crocker stacks using a spatiotemporal data set. Specifically, we ran a *flocking* simulation similar to the one run in [24] with $m = 20$ vertices

[7] *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).

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 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.

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. As depicted in Figure 5, our *moves* strategy is far more efficient than both *vineyards* and the naive `pHcol` strategies.

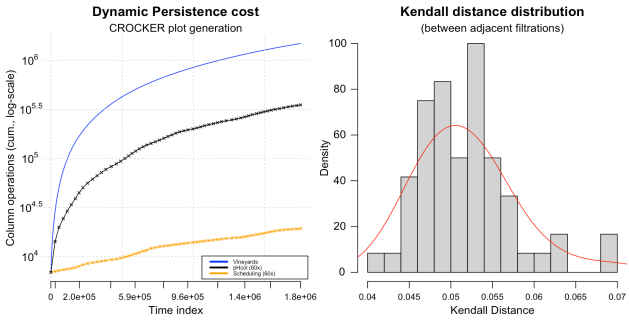


Figure 5: One the left, the cumulative number of column operations (log-scale) of the three baseline approaches tested. On the right, the normalized K_τ between adjacent filtrations depicts the coarseness of the discretization—about 5% of the $\approx O(m^2)$ simplex pairs between adjacent filtrations are discordant.

4.3 Multiparameter persistence

Given a procedure to filter a space in multiple dimensions simultaneously, a *multifiltration*, the goal of multi-parameter 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 [38]. Indeed, one of the drawbacks of persistence is its instability with respect to strong outliers, which can obscure the detection of significant topological structures [39]. One exemplary use case of multi-parameter persistence is to detect these strong outliers by filtering the data with respect to both the original filter function *and* density. In this section, we show the utility of scheduling with a real-world use case: detecting the presence of a low-dimensional topological space which well-approximates the distribution of natural images. As a quick outline, in what follows we briefly recall the fibered barcode invariant 4.3.1, summarize its potential application to a particular data set with known topological structure 4.3.2, and conclude with experiments of demonstrating how scheduling enables such applications 4.3.3.

4.3.1 Fibered barcode

Unfortunately, unlike the one-parameter case, there is no complete discrete invariant for multi-parameter persistence. Circumventing this, Lesnick et al [10] associate a variety of incomplete invariants to 2-parameter persistence modules; we focus here on the *fibered barcode* invariant, defined as follows:

Definition 3 (Fibered barcode)**.** *The fibered barcode $\mathcal{B}(M)$ of a 2D persistence module M is the map which sends each line $L \subset \mathbb{R}^2$ with non-negative slope to the barcode $\mathcal{B}_L(M)$:*

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

Equivalently, $\mathcal{B}(M)$ 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 an intuitive invariant, it is not clear how one might go about computing $\mathcal{B}(M)$ efficiently. One obvious choice is fix L via a linear combination of two filter functions, restrict M to L, and compute the associated 1-parameter barcode. However, this is an $O(m^3)$ time computation, which is prohibitive for interactive data analysis purposes.

Utilizing the equivalence between the rank and fibered barcode invariants, Lesnick and Wright [10] developed an elegant way of computing $\mathcal{B}(M)$ via a reparameterization using standard point-line duality. This clever technique effectively reduces the fibered barcode computation to a sequence of 1-D barcode computations at "template points" lying within the 2-cells of a particular planar subdivision $\mathcal{A}(M)$ of the half-plane $[0, \infty) \times \mathbb{R}$. This particular subdivision is induced by the arrangement of "critical lines" derived by the bigraded Betti numbers $\beta(M)$ of M. As the barcode of one template point $\mathcal{T}_e$ at the 2-cell $e \in \mathcal{A}(M)$ may be computed efficiently by re-using information from an adjacent template point $\mathcal{T}_{e'}$, [10] observed that computing the barcodes of all such template points (and thus, $\mathcal{B}(M)$) may be reduced to ordering the 2-cells in $\mathcal{A}(M)$ along a Eulerian path traversing the dual graph of $\mathcal{A}(M)$. The full algorithm is out of scope for this effort; we include

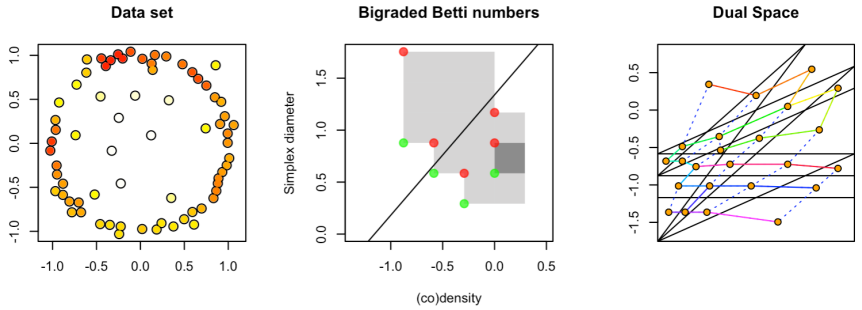


Figure 6: Bipersistence example on an 8×8 coarsened grid. On left, the input data, colored by density. In the middle, the bigraded Betti numbers $\beta_0(M)$ and $\beta_1(M)$ (green and red, respectively), the dimension function (gray), and a line L emphasizing the persistence of features with high density. On the right, the line arrangement $\mathcal{A}(M)$ lying in the dual space derived from the $\beta(M)$.

supplementary details for the curious reader in the appendix A.4.

Example 4.1: Consider a small set of noisy points distributed around S^1 containing a few strong outliers, as shown on the left side of Figure 6. Filtering this data set with respect to the Rips parameter and the complement of a kernel density estimate yields a bifiltration whose various invariants are shown in the middle figure. The gray areas indicate homology with positive dimension—the lighter gray area $\dim_1(M) = 1$ indicates a persistent loop was detected. On the right side, dual space is shown: the black lines are the critical lines that form $\mathcal{A}(M)$, the blue dashed-lines the edges of the dual graph of $\mathcal{A}(M)$, the rainbow lines overlaying the dashed-lines form the Eulerian path, and the orange barycentric points along the 2-cells of $\mathcal{A}(M)$ represent where the barcodes templates $\mathcal{T}_e$ are parameterized.

Despite its elegance, there are significant computational barriers prohibiting the 2-parameter persistence algorithm from being practical. An analysis from [10] (using *vineyards*) shows the barcodes template computation requires on the order of $O(m^3\kappa + m\kappa^2 \log \kappa)$ elementary operations and $O(m\kappa^2)$ storage, where κ is a coarseness parameter. Since 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, the scaling of the barcode template computation may approach $\approx O(m^5)$—this is both the highest complexity and most time-intensive sub-procedure the RIVET software [11] depends on. Despite this significant complexity barrier, in practice the external stability result from [40] justifies the use of a grid-like reduction procedure which approximates the module M with a smaller module M', enabling practitioners to restrict the size of κ to a relatively small constant. This in-turn dramatically reduces the size of $\mathcal{A}(M)$ and thus the number of barcode templates to compute. Moreover, the ordering of barcode templates

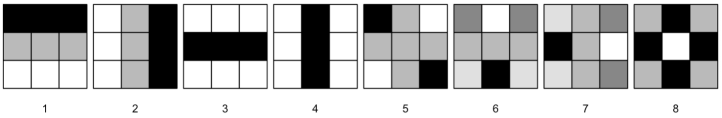
given by the dual graph traversal implies that adjacent template points should be relatively close—so long as κ is not too small—suggesting adjacent templates may productively share computations due to the high similarity of their associated filtrations. Indeed, as algorithm 3 was designed for precisely such a computation, 2-parameter persistence is prototypical of the class of methods that stand to benefit from *moves*.

4.3.2 Natural images dataset

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 the analysis by Lee et al. [41], who explored the space of high-contrast patches extracted from Hans van Hateren's [42] still image collection[8], which consists of $\approx 4,000$ monochrome images depicting various areas outside Groningen (Holland). In particular, [41] were interested in exploring how high-contrast 3×3 image patches were distributed, in pixel-space, with respect to predicted spaces and manifolds. Formally, they measured contrast using a discrete version of the scale-invariant Dirichlet semi-norm:

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

where D is a fixed matrix which upon application $x^T D x$ to an image $x \in \mathbb{R}^9$ yields a value proportional to the sum of the differences between each pixels 4 connected neighbors (given above by the relation $i \sim j$). Their research was primarily motivated by discerning whether there existed clear qualitative differences in the distributions of patches extracted from images of different modalities, such optical and range images. By mean-centering, contrast normalizing, and"whitening" the data via the Discrete Cosine Transform (DCT), they show a convenient basis for D may be obtained via an expansion of 8 certain non-constant eigenvectors, shown below:



Since these images are scale-invariant, the expansion of these basis vectors spans the 7-sphere, $S^7 \subset \mathbb{R}^8$. Using a Voronoi cell decomposition of the data, their distribution analysis suggested that the majority of data points concentrated in a few high-density regions.

In follow-up work, Carlsson et al. [43] found—using persistent homology—that the distribution of high-contrast 3×3 patches is actually well-approximated by a Klein bottle $\mathcal{M}$—around 60% of the high-contrast patches from the still image data set lie within a small neighborhood around $\mathcal{M}$ accounting for only 21% of the 7-sphere's volume. Along a similar vein, Perea [44]

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

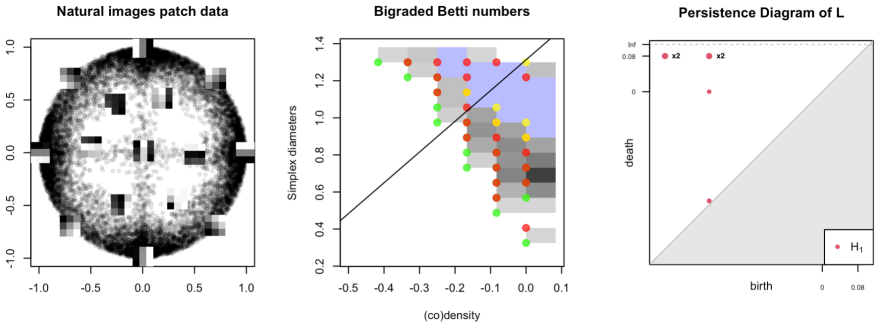


Figure 7: Bipersistence example of natural images data set on an 12×16 coarsened grid. On the left, a projection the full data set is shown, along with the 15 landmark patches. (Middle) the bigraded Betti numbers and a fixed line L over parameter space. As before, the 0/1/2 dimension bigraded Betti numbers are shown in green/red/yellow, respectively, with the blue region highlighting where $\dim(M) = 5$. (Right) five persistent features representing $B_L(M)$ are revealed from the middle, matching β_1 of the three-circle model.

established a dictionary learning framework for efficiently estimating the distribution of patches from texture images, prompting applications for persistent homology in sparse coding contexts.

If one was not aware of the analysis done by [41–44], it is not immediately clear a priori that the Klein bottle model is a good candidate for capturing the non-linearity of image patches. Indeed, armed with a refined topological intuition, Carlsson still needed to perform extensive sampling, preprocessing, and model fitting techniques in order to reveal the underlying the topological space with persistent homology [43]. One reason such preprocessing is needed is due to persistent homology's aforementioned instability with respect to strong outliers. In the ideal setting, a multi-parameter approach that accounts for the local density of points should require far less experimentation.

To demonstrate the benefit of 2-parameter persistence on the patch data, consider the (coarsened) fibered barcode computed from a standard Rips / codensity bifiltration on a representative sample of the image data from [42], shown in Figure 7. From 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 7), wherein the first Betti number is 5. Further inspection suggests one plausible candidate is the three-circle model C_3, which 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. Projecting the image data onto the first two basis vectors from the DCT shown above leads to the projection shown in the top left of Figure 7, of which 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 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, one may concur with Carlssons analysis [43] that the Klein bottle may be a reasonable candidate upon which the image data are distributed.

The degree to which multi-parameter persistence simplifies this exploratory phase cannot be understated: we believe multi-parameter persistence has a larger role to play in manifold learning. Unfortunately, as mentioned prior, the compute barriers effectively bar its use in practice.

4.3.3 Accelerating 2D persistence

Having outlined the computational theory of 2-parameter persistence, we now demonstrate the efficiency of *moves* using the same high-contrast patch data set studied in [41] by evaluating the performance of various methods at computing the fibered barcode invariant via the parameterization from A.4.

Due to the aforementioned high complexity of the fibered barcode computation, we begin by working with a subset of the image patch data $\mathcal{X}$. We combine the use of 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 the k-nearest neighbor density estimator ρ_{15} with $k = 15$. 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-nearest neighbor 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 1. We also record the number of 2-cells in $\mathcal{A}(M)$ and the number of permutations applied encountered along the traversal of the dual graph for both *vineyards* and *moves*, denoted in the table as d_K and d_{LCS}, respectively.

Table 1: Cost to computing $\mathcal{T}$ for various coarsening choices of $\beta(M)$.

$\beta(M)$	$\mathcal{A}(M)$	Col. 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, when the coarsening κ is small enough, 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`. This is further

reinforced by the observation that *vineyards* is particularly inefficient when then 1-parameter family is coarse. Indeed, *moves* requires about 3x less column operations than naively computing $\mathcal{T}$ independently. However, note that as the coarsening becomes more refined and more 2-cells are added to $\mathcal{A}(M)$, *vineyards* becomes a more viable option compared to `pHcol`—as the asymptotics suggests—though even at the highest coarsening we tested the gain in efficiency is relatively small. In contrast, *moves* 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 coarse dynamic settings. Our approach is simple, relatively easy to implement, and fully general: it does not depend on the geometry of underlying space, the choice of triangulation, or the choice of homology dimension. 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 ones discussed in section 4, such as e.g. accelerating PH featurization methods or detecting homological critical points in dynamic settings. Indeed, we see our approach as potentially useful to any situation where the structure of interest can be cast as a parameterized family of persistence diagrams. Areas of particular interest include time-series analysis and dynamic metric spaces [3].

The simple and combinatorial nature of our approach does pose some limitations to its applicability. For example, better bounds or algorithms may be obtainable if stronger assumptions can be made on how the filtration is changing with time. Moreover, if the filtration (K, f) shares little similarity to the "target" filtration (L, f'), then the overhead of reducing the simplices from $L \setminus K$ appended to the decomposition derived from K may be large enough to motivate simply computing the decomposition at L independently, especially if parallel processors are available. Our approach is primarily useful if the filtrations in the parameterized family are "nearby" in the combinatorial sense.

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 [2]. 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[9], permuting both the rows and columns generally takes at most $O(Z)$

[9] By "standard," we mean any of the common sparse representations used in scientific computing packages, like SciPy's sparse module (https://docs.scipy.org/doc/scipy/reference/sparse.html)

time, where Z is the number of non-zero entries, which can be quite expensive if the particular filtration has many cycles. As a result, the more complex sparse matrix representation from [2] 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 [14] used in the reduction algorithm to the dynamic setting. Another direction to explore would be the analysis of our approach under the cohomology computation [15], or the specialization of the move operations to specific types of filtrations such as Rips filtrations. Such adaptations may result in even larger reductions in the number of column operations, as have been observed in practice for the standard reduction algorithm [16]. Moreover, though we have carefully constructed an efficient greedy heuristic in section 3.4.2 and illustrated a different perspective with which to view our heuristic (via crossing minimization), it is an open question whether there exists a more structured reduction of (16) or (19) to a better-known problem.

6 Declarations

Ethical Approval: Not applicable.

Competing interests: The authors declare that they have no competing interests that could influence the interpretation or presentation of the research findings. There are no financial or personal relationships with individuals or organizations that could bias the outcomes of this work.

Authors' contributions: The contributions of each author to this article were as follows:

1 Matt Piekenbrock: conceptualization, methodology design, algorithm development, experimental design & analysis, software development.
2 Jose Perea: Conceptualization, methodology design, literature review, critical revision of the manuscript, final approval of the version to be published.

All authors have reviewed and approved the final version of the manuscript and have agreed to be accountable for all aspects of the work.

Funding: The research presented in this work is partially supported by the National Science Foundation through grants CCF-2006661 and CAREER award DMS-1943758. The funding source had no role in the study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Availability of data and materials: Links to the software and data used for the experiments can be found at: https://github.com/peekxc/move_schedules.

References

[1] Cohen-Steiner, D., Edelsbrunner, H., Harer, J.: Stability of persistence diagrams. Discrete & computational geometry **37**(1), 103–120 (2007)

[2] Cohen-Steiner, D., Edelsbrunner, H., Morozov, D.: Vines and vineyards by updating persistence in linear time. In: Proceedings of the Twenty-second Annual Symposium on Computational Geometry, pp. 119–126 (2006)

[3] Kim, W., Mémoli, F.: Spatiotemporal persistent homology for dynamic metric spaces. Discrete & Computational Geometry, 1–45 (2020)

[4] Morozov, D.: Persistence algorithm takes cubic time in worst case. BioGeometry News, Dept. Comput. Sci., Duke Univ **2** (2005)

[5] Polanco, L., Perea, J.A.: Adaptive template systems: Data-driven feature selection for learning with persistence diagrams. In: 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), pp. 1115–1121 (2019). IEEE

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

[7] Ulmer, M., Ziegelmeier, L., Topaz, C.M.: A topological approach to selecting models of biological experiments. PloS one **14**(3), 0213679 (2019)

[8] Zomorodian, A., Carlsson, G.: Computing persistent homology. Discrete & Computational Geometry **33**(2), 249–274 (2005)

[9] Otter, N., Porter, M.A., Tillmann, U., Grindrod, P., Harrington, H.A.: A roadmap for the computation of persistent homology. EPJ Data Science **6**, 1–38 (2017)

[10] Lesnick, M., Wright, M.: Interactive visualization of 2-d persistence modules. arXiv preprint arXiv:1512.00180 (2015)

[11] The RIVET Developers: RIVET. https://github.com/rivetTDA/rivet/

[12] Busaryev, O., Dey, T.K., Wang, Y.: Tracking a generator by persistence. Discrete Mathematics, Algorithms and Applications **2**(04), 539–552 (2010)

[13] Luo, Y., Nelson, B.J.: Accelerating iterated persistent homology computations with warm starts. arXiv preprint arXiv:2108.05022 (2021)

[14] Chen, C., Kerber, M.: Persistent homology computation with a twist. In: Proceedings 27th European Workshop on Computational Geometry, vol. 11, pp. 197–200 (2011)

[15] De Silva, V., Morozov, D., Vejdemo-Johansson, M.: Dualities in persistent (co) homology. Inverse Problems **27**(12), 124003 (2011)

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

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

[18] Attali, D., Glisse, M., Hornus, S., Lazarus, F., Morozov, D.: Persistence-sensitive simplication of functions on surfaces in linear time. In: TopoInVis' 09 (2009)

[19] Edelsbrunner, H., Letscher, D., Zomorodian, A.: Topological persistence and simplification. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 454–463 (2000). IEEE

[20] Delfinado, C.J.A., Edelsbrunner, H.: An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere. Computer Aided Geometric Design **12**(7), 771–784 (1995)

[21] Chen, C., Kerber, M.: An output-sensitive algorithm for persistent homology. Computational Geometry **46**(4), 435–447 (2013)

[22] Oesterling, P., Heine, C., Weber, G.H., Morozov, D., Scheuermann, G.: Computing and visualizing time-varying merge trees for high-dimensional data. In: Topological Methods in Data Analysis and Visualization, pp. 87–101 (2015). Springer

[23] Kapron, B.M., King, V., Mountjoy, B.: Dynamic graph connectivity in polylogarithmic worst case time. In: Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1131–1142 (2013). SIAM

[24] Topaz, C.M., Ziegelmeier, L., Halverson, T.: Topological data analysis of biological aggregation models. PloS one **10**(5), 0126383 (2015)

[25] Xian, L., Adams, H., Topaz, C.M., Ziegelmeier, L.: Capturing dynamics of time-varying data via topology. arXiv preprint arXiv:2010.05780 (2020)

[26] Boissonnat, J.-D., Snoeyink, J.: Efficient algorithms for line and curve segment intersection using restricted predicates. Computational Geometry

16(1), 35–52 (2000)

[27] Diaconis, P., Graham, R.L.: Spearman's footrule as a measure of disarray. Journal of the Royal Statistical Society: Series B (Methodological) **39**(2), 262–268 (1977)

[28] Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pp. 39–48 (2000). IEEE

[29] Labarre, A.: Lower bounding edit distances between permutations. SIAM Journal on Discrete Mathematics **27**(3), 1410–1428 (2013)

[30] Kumar, S.K., Rangan, C.P.: A linear space algorithm for the lcs problem. Acta Informatica **24**(3), 353–362 (1987)

[31] Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for lcs and other sequence similarity measures. In: 2015 IEEE 56th Annual Symposium on Foundations of Computer Science, pp. 59–78 (2015). IEEE

[32] Bespamyatnikh, S., Segal, M.: Enumerating longest increasing subsequences and patience sorting. Information Processing Letters **76**(1-2), 7–11 (2000)

[33] Baik, J., Deift, P., Johansson, K.: On the distribution of the length of the longest increasing subsequence of random permutations. Journal of the American Mathematical Society **12**(4), 1119–1178 (1999)

[34] Boissonnat, J.-D., CS, K.: An efficient representation for filtrations of simplicial complexes. ACM Transactions on Algorithms (TALG) **14**(4), 1–21 (2018)

[35] Biedl, T., Brandenburg, F.J., Deng, X.: On the complexity of crossings in permutations. Discrete Mathematics **309**(7), 1813–1823 (2009)

[36] Dinu, L.P., Manea, F.: An efficient approach for the rank aggregation problem. Theoretical Computer Science **359**(1-3), 455–461 (2006)

[37] Blelloch, G.E., Reid-Miller, M.: Fast set operations using treaps. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 16–26 (1998)

[38] Lesnick, M.P.: Multidimensional interleavings and applications to topological inference. PhD thesis, Stanford University (2012)

[39] Buchet, M., Chazal, F., Dey, T.K., Fan, F., Oudot, S.Y., Wang, Y.: Topological analysis of scalar fields with outliers. In: 31st International

Symposium on Computational Geometry, pp. 827–841 (2015). Schloss Dagstuhl, Leibniz-Zentrum fü Informatik GmbH

[40] Landi, C.: The rank invariant stability via interleavings. arXiv preprint arXiv:1412.3374 (2014)

[41] Lee, A.B., Pedersen, K.S., Mumford, D.: The nonlinear statistics of high-contrast patches in natural images. International Journal of Computer Vision **54**(1), 83–103 (2003)

[42] Hateren, J.H.v., Schaaf, A.v.d.: Independent component filters of natural images compared with simple cells in primary visual cortex. Proceedings: Biological Sciences **265**(1394), 359–366 (1998)

[43] Carlsson, G., Ishkhanov, T., De Silva, V., Zomorodian, A.: On the local behavior of spaces of natural images. International journal of computer vision **76**(1), 1–12 (2008)

[44] Perea, J.A., Carlsson, G.: A klein-bottle-based dictionary for texture representation. International journal of computer vision **107**(1), 75–97 (2014)

[45] Carlsson, G., Zomorodian, A.: The theory of multidimensional persistence. Discrete & Computational Geometry **42**(1), 71–93 (2009)

[46] Lesnick, M., Wright, M.: Computing minimal presentations and bigraded betti numbers of 2-parameter persistent homology. arXiv preprint arXiv:1902.05708 (2019)

[47] Boissonnat, J.-D., Preparata, F.P.: Robust plane sweep for intersecting segments. SIAM Journal on Computing **29**(5), 1401–1421 (2000)

[48] Christofides, N.: Worst-case analysis of a new heuristic for the travelling salesman problem. In: Operations Research Forum, vol. 3, pp. 1–4 (2022). Springer

A Appendix

A.1 Algorithms

A.1.1 Reduction Algorithm

The reduction algorithm, also called the "standard algorithm," is the most often used modality for computing persistence. While there exists other algorithms for computing persistence, they are typically not competitive with the reduction algorithm in practice. We outline the reduction algorithm below in Algorithm 4. The algorithm begins by copying D to a new matrix R, to be

Algorithm 4 Reduction Algorithm (`pHcol`)

Require: $D = (m \times m)$ filtration boundary matrix

Ensure: R is reduced, V is full rank upper triangular, and $R = DV$

1: **function** REDUCTION(D)
2: $(R, V) \leftarrow (D, I)$
3: **for** $j = 1$ **to** m **do**
4: **while** $\exists\, i < j$ **such that** $\text{low}_R(i) = \text{low}_R(j)$ **do**
5: $\lambda \leftarrow \text{pivot}_R(j)/\text{pivot}_R(i)$
6: $(\text{col}_R(j), \text{col}_V(j)) \mathrel{-}= (\lambda \cdot \text{col}_R(i), \lambda \cdot \text{col}_V(i))$
7: **return** (R, V)

subsequently modified in-place. After setting V is set to the identity, the algorithm proceeds with column operations on both R and V, left to right, until the decomposition invariants are satisfied. Since each column operation takes $O(m)$ and there are potentially $O(k)$ columns in D with identical low entries (line 4 in 4, observe the reduction algorithm below clearly takes $O(m^2k)$ time. Since there exists complexes where $k \sim O(m)$, one concludes the bound of $O(m^3)$ is tight [4], though this seems to only be true on pathological inputs. Indeed, a more refined analysis by Edelsbrunner et al. [19] shows the reduction algorithm scales by the sum of squares of the cycle persistences, which is an output-sensitive bound.

Move Algorithms

As we've covered the moves algorithm extensively in section 2.3, we now record the algorithmic components of both *MoveRight* and *MoveLeft*. Though conceptually similar, note that 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 [12] 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, though R stays reduced during this cancellation procedure in all but one column, the subsequent permutation to R requires reducing a pair of columns which may cascade into a larger chain of column operations to keep R reduced. This is due to the fact that higher entries in columns in R (above the pivot entry) may very well introduce additional non-reduced columns after R is permuted. 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, like move right, we can still separate the algorithm into a reduction and restoration phase—see Algorithm 2. Moreover, since R is reduced in all but one column by line 6 in Algorithm 2, we can still guarantee the number of low entries to reduce in R will be at most $|i - j|$. For a supplementary description of the move algorithm, see [12].

A.1.2 LCS-Sort

Here we record explicitly the schedule construction algorithm outlined in section 3.3. The algorithm is simple enough to derive using the rules discussed in section 3.3 (namely, equation (14), but nonetheless for posterity sake we record it here for the curious reader; the pseudocode is given in Algorithm 5.

The high level idea of the algorithm is to first construct the LCS between two permutations $p, q \in S_m$. To do this efficiently, one re-labels $q \mapsto \iota$ to the identity permutation $\iota = [m]$ and applies a consistent re-labeling $p \mapsto \bar{p}$. This relabeling preserves the LCS distance and has the additional advantage that $\bar{q} = \iota = [m]$ is a strictly increasing subsequence, and thus computing the LCS between $p, q \in S_m$ reduces to computing the LIS $\mathcal{L}$ of $\bar{p}$. By sorting $\bar{p} \mapsto \iota$ via operations which (strictly) increase the size of $\mathcal{L}$, we ensure that the size of the set of corresponding schedule is exactly $m - |\mathcal{L}|$.

Algorithm 5 Schedule construction algorithm

Require: Fixed $\bar{p} \in S_m$, LIS $\mathcal{L}$ of $\bar{p}$, and heuristic h
Ensure: $[m] = s_d \circ s_{d-1} \circ \cdots \circ s_1 \circ \bar{p}$ for output sequence $\mathcal{S} = (s_i)_{i=1}^{d}$,

1: **function** SCHEDULE($\bar{p}$, $\mathcal{L}$, $h =$ `greedy`) ▷ See 3.4.1 for heuristic discussion
2: $(\mathcal{S}, \mathcal{D}) \leftarrow (\emptyset, [m] \setminus \mathcal{L})$
3: **while** $\mathcal{D}$ is not empty **do**
4: Select an element $k \in \mathcal{D}$ using heuristic h ▷ e.g. equation (20)
5: $\mathrm{k}_{\mathrm{pred}} \leftarrow \max\{\ell \in \mathcal{L} \mid \ell \leq k\}$ ▷ $O(\log\log m)$ using $\mathcal{U}$
6: $\mathrm{k}_{\mathrm{succ}} \leftarrow \min\{\ell \in \mathcal{L} \mid \ell \geq k\}$ ▷ $O(\log\log m)$ using $\mathcal{U}$
7: $(i, i_p, i_n) \leftarrow (\bar{p}^{-1}(k), \bar{p}^{-1}(\mathrm{k}_{\mathrm{pred}}), \bar{p}^{-1}(\mathrm{k}_{\mathrm{succ}}))$ ▷ $O(1)$
8: $j \leftarrow$ arbitrary $j \in [i_p, i_n)$ **if** $i < i_p$ **else** $j \in (i_p, i_n]$ ▷ $O(1)$
9: $(\mathcal{S}, \mathcal{D}, \mathcal{L}) \leftarrow (\mathcal{S} \cup (i, j), \mathcal{D} \setminus k, \mathcal{L} \cup k)$ ▷ $O(\log\log m)$
10: $\bar{p}^{-1} \leftarrow \bar{p}^{-1} \circ m_{ij}^{-1}$ where m_{ij} is given by (10) ▷ $O(m)$
11: **return** $\mathcal{S}$

The algorithmic steps are as follows: given a LIS $\mathcal{L}$ has been computed from $\bar{p}$, since $\mathcal{L}$ is strictly increasing, the only elements left to permute are in $\mathcal{L} \setminus \bar{p}$, which we denote with $\mathcal{D}$. After choosing any $\sigma \in \mathcal{D}$, one then applies a cyclic permutation to $\bar{p}$ that moves σ into any position that increases the size of $\mathcal{L}$. To do this efficiently, we use a set-like data structure $\mathcal{U}$ that supports efficient querying the successor and predecessor of any given $s \in \bar{p}$ with respect to $\mathcal{L}$ (such as a vEB tree). The simplified pseudocode also uses the inverse permutation $\bar{p}$ to query the position of a given element $\sigma \in \bar{p}$, though a more efficient representation can be used based off of an implicit treap of the *displacements* array could also be used, see section 3.4.2. After σ is inserted into $\mathcal{L}$, we update $\bar{p}$, its inverse permutations $\bar{p}^{-1}$, $\mathcal{D}$ and $\mathcal{T}$ prior to the next move. The final set of permutations which sort $\bar{p} \to \iota$ (or equivalently, $p \mapsto q$) are stored in an array $\mathcal{S}$, which is then returned for further use.

A.2 Greedy Counter Example

In this section we give a simple counter-example showing that the strategy that greedily chooses valid move permutations $\{m_{ij}\}$ minimizing the quantity

$$\text{cost}_{RV}(m_{ij}) = \sum_{l=i+1}^{j} \mathbb{1}\left(v_l(i) \neq 0\right) + \sum_{l=1}^{m} \mathbb{1}\left(\text{low}_R(l) \in [i,j] \text{ and } r_l(i) \neq 0\right)$$

can lead to arbitrarily bad behavior. A pair of filtrations is given below, each comprising the 1-skeleton of a 3-simplex. Relabeling (K, f) to the index set $f : K \to [m]$ and modifying (K, f') accordingly yields the permutations:

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

$$(K, f') = \{a\ b\ c\ d\ x\ y\ z\ u\ v\ w\} = 1\ 2\ 3\ 4\ 8\ 9\ 10\ 5\ 6\ 7$$

The values colored in red corresponds to $\text{LCS}(f, f')$. For this example, the edit distance is $d = m - |\text{LCS}(f, f')|$ implies exactly 3 moves are needed to map $f \mapsto f'$. There are six possible valid schedules of moves:

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

where the notation m_{xy} represents the move permutation that moves x to the position of y. The cost of each move operation and each schedule is recorded in Table 2. Note the greedy strategy which always selects the cheapest move in succession 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 on the first iteration. While the cheapest schedule S_1 is in this candidate set, an iterative greedy procedure would pick either S_2 or S_5, depending on the tie-breaker—thus, a greedy approach picking the lowest-cost move may not yield an optimal schedule. Indeed, as the most expensive schedule S_6 is in initial iterations

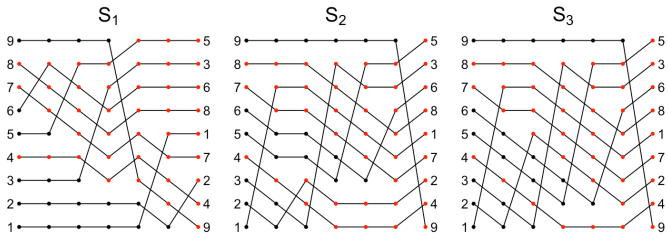
Table 2: Move schedule costs

	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

candidate set, we see that a greedy-procedure with an arbitrary tie-breaker could potentially yield a *maximal-cost* schedule.

A.3 Crossing minimization

Conceptually, one way to view (17) is as a crossing minimization problem over a set of $k-1$ bipartite graphs. To see this, consider two permutations: p and $m_{ij} \circ p$, where m_{ij} is a move permutation. Drawing $(p, m_{ij} \circ p)$ as a bipartite graph (U, V, E), observe that there are $|i-j|$ edge crossings in the graph, and thus minimizing (17) is akin to a structured variation of the k-layered crossing minimization problem. We give an example visually relating the problem of minimizing *net displacement* to the problem of minimizing crossing in a k-layered bipartite graph. Let $p = (1\,2\,3\,4\,5\,6\,7\,8\,9)$ and $q = (9\,4\,2\,7\,1\,8\,6\,3\,5)$. An example of three possible schedules, S_1, S_2, and S_3 sorting p into q is given in the figure below. Each column represents the successive application



of a move m_{ij} in the schedule, and the edges track the movement of each element of the permutation. Black/red vertices correspond to elements inside and outside the LCS, respectively. All three schedules were generated from the same $\mathrm{LCS}(p, q) = (4\,7\,8)$ and each schedule transforms $p \mapsto q$ in $d = 6$ moves. In this example, S_1 matches the minimal number of crossings amongst all possible schedules, since $K_\tau(p, q) = 21$.

A.4 2-parameter persistence

We now describe the reparameterization between the bigraded Betti numbers and the set of "critical lines" Lesnick and Wright [38] used to create their

interactive 2d persistence algorithm, beginning with 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 with non-negative finite slope, and $\mathcal{L}^\circ$ the collection of all affine lines with positive finite slope. Define the *line* and *point* dual transforms $\mathcal{D}_\ell$ and $\mathcal{D}_p$, respectively, as follows:

$$\begin{aligned} \mathcal{D}_\ell : \mathcal{L} &\to [0,\infty) \times \mathbb{R} & \qquad \mathcal{D}_p : [0,\infty) \times \mathbb{R} &\to \mathcal{L} \\ y = ax + b &\mapsto (a, -b) & (c,d) &\mapsto y = cx - d \end{aligned} \tag{21}$$

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:

$$\text{push}_L(a) \mapsto \min\{v \in L \mid a \leq v\} \tag{22}$$

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 an ordered partition S_L on S

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

Definition 4 (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 $[0,\infty) \times \mathbb{R}$. This planar subdivision, denoted by $\mathcal{A}(M)$, is thus entirely determined by S under point line duality. A corollary from [10] shows 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 simply 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.

To connect this construction back to persistence, one requires the definition of bigraded Betti numbers. For our purposes, the i^{th}-graded Betti number of M is simply a function $\beta_i(M) : \mathbb{R}^2 \to \mathbb{N}$ whose values indicate the the number of elements at each degree in a basis of the i^{th} module in a free resolution for M—the interested reader is referred to [10, 45] for a more precise algebraic definition. 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 [10] 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:

$$\mathcal{B}_L(M) = \{[\,\mathrm{push}_L(a),\, \mathrm{push}_L(b)\,) \mid (a,b) \in \mathcal{T}^e, \mathrm{push}_L(a) < \mathrm{push}_L(b)\} \quad (23)$$

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.

A.4.1 Invariant computation

Computationally, the algorithm from [46] can be summarized into three steps:

1 Compute the bigraded Betti numbers $\beta(M)$ of M
2 Construct a line arrangement $\mathcal{A}(M)$ induced by critical lines from (1)
3 Augment $\mathcal{A}(M)$ with *barcode templates* $\mathcal{T}_e$ at every 2-cell $e \in \mathcal{A}(M)$

Computing (1) takes approximately $\approx O(m^3)$ using a matrix algorithm similar to Algorithm 4 [46]. 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 [47]. In terms of space 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 $\beta(M)$.

There are several approaches one can use to compute $\mathcal{T}$, the simplest being to run Algorithm 4 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 [10] 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 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 which is proportional to the computational disparity between adjacent template computations. The optimal path Γ^* that minimizes the computation time is then the minimal length path 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 [48], and thus can be used so long as the distance function between templates is a valid metrics. [38] use the Kendall distance between the

push-map induced filtrations, but other options are available—for example, any of the combinatorial metrics we studied in Section 3.4.