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

The standard procedure for computing the persistent homology of a filtered simplicial complex with $\boldsymbol{m}$ simplices is the reduction algorithm. 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. Computationally, simulating 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 that requires only $\boldsymbol{O(m \log\log m)}$ time and $\boldsymbol{O(m)}$ space to construct. By reduction to a particular longest common subsequence problem, we show the storage needed to employ this strategy is actually sublinear in expectation. Exploiting this connection, we show experimentally 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.

Keywords: Computational topology, Persistent homology, Topological data analysis

MSC Classification: 68T09 , 55N31 , 62R40

1 Introduction

1.1 Overview

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. With state of the art tools, there is no recourse when faced with a time-varying complex containing millions of simplices across thousands of snapshots in time. Moreover, acquiring such a capability has far-reaching consequences: methods that vectorize persistence diagrams for machine learning purposes all immediately become computationally viable tools in dynamic settings. Persistence-based examples 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 diagram of a fixed filtration K with m simplices in $O(m^3)$ time via a decomposition $R = DV$ (or $RU = D$) of the boundary matrix D of K. In particular, the vineyards algorithm 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. 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 $d >> m$, this $O(md)$ approach is far more efficient than the $O(dm^3)$ "naive" strategy of computing the diagrams at every time point independently.

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

[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.

perhaps explained by Lesnick and Wright [10]: "While an update to an RU decomposition involving few transpositions is very fast in practice... many transpositions can be quite slow... it is sometimes much faster to simply recompute the RU-decomposition from scratch using the standard persistence algorithm." Indeed, they observed maintaining the decomposition along a certain parameterized family is the most computationally demanding aspect of RIVET [11], software for computing and visualizing two-parameter persistent homology.

This work seeks to further understand and remedy this discrepancy: building on the work presented in [12], we introduce a coarser approach to the vineyards algorithm. While vineyards is designed for handling 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]

1.2 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 similar in spirit, 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

[3] Code is available here: https://github.com/peekxc/dart

the 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 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 et al. [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 ($p = 0$) is the only dimension of interest, computing and updating both the persistence and rank information is greatly simplified. For example, if the relations are available a-priori, obtaining a tree representation fully characterizing the connectivity of the underlying space (also known as the *incremental connectivity* problem) takes just $O(\alpha(n)n)$ time using the disjoint-set data structure, where $\alpha(n)$ is the extremely slow-growing inverse Ackermann function. Adapting this approach to the time-varying setting, Oesterling et al. [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].

1.3 A Motivating Example

We illustrate why the vineyards algorithm does not always yield an efficient strategy for time-varying setting with a simple experiment. 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:

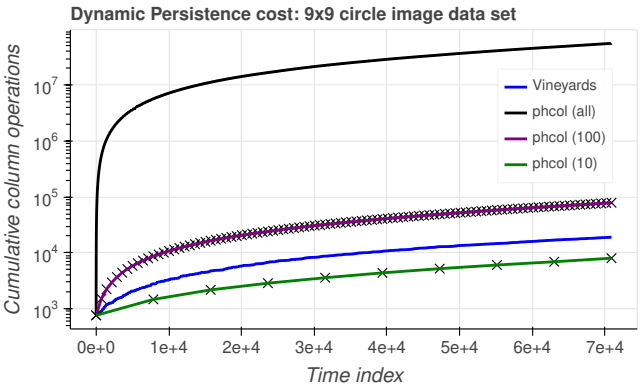
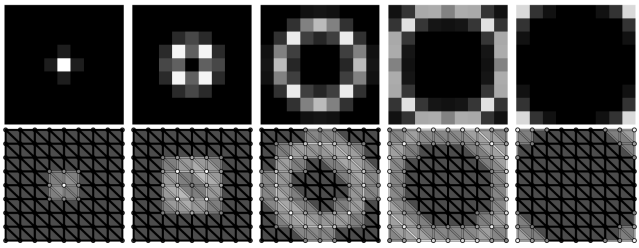


Figure 1: The cumulative column operations needed to compute persistence across the time-varying filtration of grayscale images. Observe 10 independent persistence computations evenly spaced in time (green line) captures the major topological changes and is the most computationally efficient approach shown.



Each image contains a fixed number of pixels whose intensities vary with time. For each image, we build a fixed-size filtered simplicial complex using the *Freudenthal* triangulation of the plane whose inclusion maps are obtained by lower stars of pixel values. Two events critically change the pairing function: the first occurs when the central connected component splits to form a cycle, and the second when 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 2). Suppose a practitioner wanted to know the major homological changes a time-varying filtration encounters over time. Since it is unknown a priori at 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 of 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 only not guaranteed to capture all homological changes in persistence that occur in simulating a continuous 1-parameter family of diagrams.

The cumulative cost (in total column operations) of these various approaches are shown on the left side of Figure 1, wherein the reduction algorithm (`pHcol`) and the vineyards algorithm are compared. Two discrete strategies (green and purple) and two continuous strategies (black and blue) are shown. Note that without knowing where 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 pairing function is unchanged. Indeed, as shown by the case where $n = 10$, applying `pHcol` independently between relatively "close" 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—in terms of the number of valid intermediate decomposition states, as compared to vineyards—for dynamic persistence. We give a tight lower bound on the number of moves needed to perform an arbitrary permutation to the $R = DV$ decomposition, and give a proof of optimality by a reduction to the permutation edit distance problem. We also give worst-case sizes of these quantities in expectation as well as efficient algorithms for achieving these bounds—both of which are derived from a reduction to the Longest Increasing Subsequence (LIS) problem. This reduction yields an efficient algorithm for generating sequences of moves $(s_1, s_2, \ldots, s_d)$ 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 approximately 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.

1.4 Main results

Given a simplicial complex K with filtration function $f : K \to [m]$, 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, 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 $D' = P^T DP$, such an update scheme has the form:

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

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 (and none of the intermediate $d-1$ decompositions), updating $R \mapsto R'$ using the approach from [2] matches the complexity of computing $R' = D'V'$ independently.

We now summarize the main results obtained in this effort (Theorem 1): suppose one has a sequence of permutations $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ yielding a corresponding sequence of decompositions:

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

where each permutation s_i is a particular type of cyclic permutation (see section 3.2). If $i_* < j_*$, our first result extends [12] by showing that (1) can be computed in using $O(\kappa)$ column operations, where:

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

where 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 take advantage of in Section 3.4.

Our second result turns towards upper 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 the *smallest* such sequence that maps $f \mapsto f'$ has size d (Proposition 3), where:

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

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. Although d clearly is $O(m)$ for pathological inputs, we give evidence that $d \sim m - \sqrt{m}$ in expectation for random filtrations is not too pessimistic for truly random filtrations (Corollary 1), and we give empirical results suggesting d is tends to be much smaller for time-varying filtrations.

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 our move schedules and provide efficient algorithms to make this alternative viable. 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 an simplexwise filtration via a set of *condensing*, *refining*, and *reindexing* maps (see [16] for more details). Equivalently, we can also define a filtration as a pair (K, f) where K is a simplicial complex and $f : K \to I$ is a *filter function* over a totally ordered index set I satisfying $f(\tau) \leq f(\sigma)$ whenever $\tau \subseteq \sigma$, for any $\tau, \sigma \in K$. 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 $\{K_i\}_{i\in[m]}$ is a filtration, then the inclusion maps $K_i \subset K_{i+1}$ 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 $\mathrm{dgm}_p(K)$, and referred to as the p-th *persistence diagram* of K. If a homology class is born at time i and dies entering time j, the difference $|i - j|$ is called the *persistence* of that class. In practice, filtrations often arise from triangulations parameterized by geometric scaling parameters, and the "persistence" of a homology class actually refers to its lifetime with respect to the scaling parameter.

Let $\mathbb{X}$ be a triangulable topological space and K an abstract simplicial complex 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 2 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 main output of the reduction algorithm is a matrix decomposition $R = DV$, where the persistence diagram is encoded in R and the generating cycles in the columns of V. To begin the reduction, one starts by assembling 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

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 $\text{low}_R(i)$ denotes the largest row index of a non-zero entry in column i of R. We call the decomposition satisfying these three invariants *valid*. Note that though the matrices R and V are not unique, the collection of persistent pairings are [8]. The persistence diagram of the corresponding filtration K may be read off from lowest entries in R, once R is in reduced form.

It is at times more succinct to restrict to specific sub-matrices D based on the homology dimension p, and so we write D_p represent the $d_{p-1} \times d_p$ matrix representing the linear operator ∂_p (the same notation is extended to R and V). We illustrate the reduction algorithm with an example pair (R_1, V_1) below.

Example 2.1: Reduction 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 $K = (u, v, w, a, b, c)$. Using $\mathbb{Z}_2$ coefficients for simplicity, the reduction proceeds to compute $H_1(K)$ as follows:

$$
\begin{array}{c} D_1 \\ \\ u \\ v \\ w \end{array}
\begin{array}{c} a\ b\ c \\ \begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ 1 & \underline{1} & \end{bmatrix} \end{array}
,\quad
\begin{array}{c} I_1 \\ \\ a \\ b \\ c \end{array}
\begin{array}{c} a\ b\ c \\ \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \end{array}
\rightarrow
\begin{array}{c} \\ \\ u \\ v \\ w \end{array}
\begin{array}{c} a\ b\ c \\ \begin{bmatrix} 1 & 1 & 1 \\ & 1 & \underline{1} \\ 1 & & \end{bmatrix} \end{array}
,\quad
\begin{array}{c} \\ \\ a \\ b \\ c \end{array}
\begin{array}{c} a\ b\ c \\ \begin{bmatrix} 1 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} \end{array}
\rightarrow
\begin{array}{c} R_1 \\ \\ u \\ v \\ w \end{array}
\begin{array}{c} a\ b\ c \\ \begin{bmatrix} 1 & 1 & \\ & 1 & \\ 1 & & \end{bmatrix} \end{array}
,\quad
\begin{array}{c} V_1 \\ \\ b \\ a \\ c \end{array}
\begin{array}{c} a\ b\ c \\ \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
$$

Since column c in R_1 is 0, the chain $d_1 + d_2 + d_3$ 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, the number of column operations has been observed to be super-linear in practice, due impart to the high sparsity and structure of D. Moreover, many variations and optimizations to Algorithm 2 have been proposed over the past decade, see [14, 16, 17] for an overview.

2.2 Vineyards

Consider a homotopy $F(x, \tau) : \mathbb{X} \times [0, 1] \rightarrow \mathbb{R}$ on a triangulable topological space $\mathbb{X}$, and denote its "snapshot" at a given time-point τ by $f_\tau(x) = F(x, \tau)$. The snapshot f_0 denotes the initial function at time $\tau = 0$ and f_1 denotes the function at the last time step. As τ varies in $[0, 1]$, the points in $\text{dgm}_p(f_\tau)$ trace curves in $\mathbb{R}^3$, which will be continuous if F is continuous and the f_τ's are tame, due to the stability of persistence. Cohen-steiner et al. [1] referred to these curves as *vines*, as collection of which forms as *vineyard*—the geometric

analogy meant to act as a guidepost for practitioners seeking to understand how subtle changes occurring to the topological structure over time.

The original purpose of the vineyards algorithm, 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 all time points interleaving 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 transpositions is sufficient to simulate persistence dynamically.

These rules prescribe certain column and row operations to apply to a given matrix decomposition either before, during, or after each transposition to ensure the decomposition is valid. Formally, let S_i^j represent the upper-triangular matrix such that AS_i^j is equivalent to the elementary operation that adds column i of A to column j of A and $S_i^j A$ is equivalent to adding row j of A to row i of A. Similarly, let P denote the matrix so that AP applies a fixed permutation to the columns of A and $P^T A$ applied the same permutation to the rows of A. Since the columns of P are orthonormal, then $P^{-1} = P^T$, and thus one would write $P^T AP$ to denote the application of 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 on which vineyards works, 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.

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

$$
\begin{array}{c} V_1 \\ a \\ b \\ c \end{array}\!\!\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & \underline{1} & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\xrightarrow{S_1^2}
\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\xrightarrow{P}
\begin{array}{c} \begin{array}{ccc} b & a & c \end{array} \\ \begin{bmatrix} 1 & & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
\xrightarrow{S_1^2}
\begin{array}{c} \\ b \\ a \\ c \end{array}\!\!\begin{array}{c} \begin{array}{ccc} b & a & c \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{bmatrix} \end{array}
$$

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 the S_1^2 on the right to R_1 and V_1. After this operation, the permutation may be safely applied to V_1. 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, a host of other factors affect the runtime efficiency of the vineyards algorithm. On the positive side, as both the V and R matrices tend to be quite sparse, the cost of a given column operation is proportional the number of non-zero field coefficients in the two columns being modified. As a rule of thumb, 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 Move(i, j) is a set of rules for maintaining a valid decomposition under the application of a permutation P that moves a simplex σ_i at position i to position j. If $j = i \pm 1$, this operation is an adjacent transposition, and thus in some sense moves generalizes vineyards. However, despite being conceptually equivalent to performing a sequence of contiguous transpositions, the move operation presented by Busaryev exhibits several attractive qualities distinct from the vineyards approach that warrants further study.

For completeness, we recapitulate the motivation of the moves algorithm from [12]. Let K denote a filtration of size $m = |K|$ and $R = DV$ its decomposition, where $R = [r_1, r_2, \ldots, r_m]$ and $V = [v_1, v_2, \ldots, v_m]$ denote the columns of R and V, respectively. By definition, if $r_j = Dv_j = 0$, then the inclusion $K_{j-1} \hookrightarrow K_j$ introduced a new cycle whose generator is given by:

$$r_j = Dv_j = \alpha_i^{(i)} \cdot \partial(\sigma_i) + \alpha_i^{(i+1)} \cdot \partial(\sigma_{i+1}) + \cdots + \alpha_i^{(j)} \cdot \partial(\sigma_j) = 0 \qquad (5)$$

Now, 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, the non-zero coefficients from (5) must be set to zero, otherwise $\partial(\sigma_i)$ contributes to boundary chains $r_{i+1}, \ldots, r_j$ earlier in the filtration. As these coefficients are recorded by v_j, setting $\alpha_i^k = 0$ amounts to setting the row entries $v_k(i) = 0$ for all $k \in [i + 1, j]$—these cancellations manifest as column operations $(*)$ used to ensure $PV(*)$ is upper-triangular, thus maintaining

invariant I2. Notice that these operations may induce an unreduced $R' = PR(*)P^T$, breaking invariant I3. We could reduce R' with an additional k operations, however its possible that $k \sim O(|i-j|^2)$, rendering the efficiency of this approach at best that of vineyards and at worse that of the standard algorithm.

To bypass this difficulty, Busaryev et al. observed that since the initial R is reduced, if it contains s pivot entries in the columns $[i, j]$ of R, then R' must also have s pivots. In particular, if during the cancellation of $v_{i+1}(i) = v_{i+2}(i) = 0$ the column $r_{i+2} \mapsto r'_{i+2}$ becomes unreduced, then the pivot $\text{low}_R(i+2)$ becomes a *free*, possibly becoming a pivot again in $r'_{i+3}, \ldots, r'_j$. Thus, if r_{i+2} is copied prior to modification to a *donor column*, it may re-use or *donate* its pivot entry to a later column $r_{i+3}, \ldots, r_j$. Repeating this process at most $j - i - 1$ times ensures R' stays reduced in all except possibly for the i-th column by the end of the movement—and since the k-th such operation simultaneously sets $v_k(i) = 0$ without creating non-zeros at indices $v_k(j)$ for $j > k$, 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.

$$
\begin{array}{c}
\begin{array}{cc} d_R & a \\ u & 1 \\ v & \\ w & 1 \end{array}
\begin{array}{cc} R & a\ b\ c \\ u & 1\ 1 \\ v & \ \ 1 \\ w & 1 \end{array}
\rightarrow
\begin{array}{cc} & b \\ & 1 \\ & 1 \end{array}
\begin{array}{c} a\ b\ c \\ 1 \\ 1\ \underline{1} \end{array}
\rightarrow
\begin{array}{cc} & c \\ & \\ & \end{array}
\begin{array}{c} a\ b\ c \\ 1\ \ 1 \\ \ \ 1\ 1 \\ 1\ \underline{1} \end{array}
\xrightarrow{P}
\begin{array}{cc} & a \\ & \\ & \end{array}
\begin{array}{c} b\ c\ a \\ \ \ 1\ 1 \\ 1\ 1 \\ 1\ \ \underline{1} \end{array}
\xrightarrow{d_R}
\begin{array}{c} b\ c\ a \\ \ \ 1 \\ 1\ 1 \\ 1 \end{array}
\\[2ex]
\begin{array}{cc} d_V & a \\ a & 1 \\ b & \\ c & \end{array}
\begin{array}{cc} V & a\ b\ c \\ a & 1\ \underline{1}\ \underline{1} \\ b & \ \ 1\ \ 1 \\ c & \ \ \ \ 1 \end{array}
\rightarrow
\begin{array}{cc} & b \\ & 1 \\ & 1 \end{array}
\begin{array}{c} a\ b\ c \\ 1\ \ \underline{1} \\ \ \ 1\ \underline{1} \\ \ \ \ \ 1 \end{array}
\rightarrow
\begin{array}{cc} & c \\ & 1 \\ & 1 \\ & 1 \end{array}
\begin{array}{c} a\ b\ c \\ 1 \\ \ \ 1 \\ \ \ \ \ 1 \end{array}
\xrightarrow{P}
\begin{array}{cc} & a \\ & 1 \\ & 1 \\ & 1 \end{array}
\begin{array}{c} b\ c\ a \\ 1 \\ \ \ 1 \\ \ \ \ \ 1 \end{array}
\xrightarrow{d_V}
\begin{array}{c} b\ c\ a \\ 1\ \ 1 \\ \ \ 1\ 1 \\ \ \ \ \ 1 \end{array}
\end{array}
$$

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 3 and for *MoveLeft* in Algorithm 4.

As shown by example 2.3, move updates are cheaper than vineyard updates when $j > i + 1$. More formally, we recall the proposition shown in [12]:

Proposition 1 (Busaryev et al. [12])**.** *Let K denote a filtration with n simplices of dimension $p-1$, p, and $p+1$, respectively, and 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, including the two computationally attractive properties given below:

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

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

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

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 $P^T V P$ non-upper triangular, to maintain invariant I2. Using vineyards, one must consistently perform $|i-j|-1$ non-zero status queries interleaved between repairing column operation. A move operation, on the other hand, groups these status queries into a single pass prior to performing any modifying operations.

Property M2 implies 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 order in K' via insertion sort. Note that the number of valid decomposition produced by vineyards is bounded above by $O(m^2)$ if each pair of simplices $\sigma_i, \sigma_j \in K$ switches its relative ordering at most once during the interpolation from K to K'.

As a final note, we remark the combination of *MoveRight* and *MoveLeft* enable the addition or deletion of simplices to the underlying complex. In particular, given K and its 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*$(m + 1, i)$, where $i = f'(\sigma)$. 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

Let us begin with a brief overview of the pipeline, outlined in Algorithm 1 below. As before, we assume as input a discrete 1-parameter family of filtrations $\mathcal{K} = (K_1, K_2, \ldots, K_n)$ of an abstract simplicial complex K with $|K| = m$, and the goal being to compute their persistence diagrams. Fix reindexing bijections $\rho_i : K_i \to K_{i+1}$. Each ρ_i induces a bijection $\rho_i^* : [m] \to [m]$, or equivalently, a permutation of the index set $[m]$. For each pair of filtrations (K_i, K_{i+1}), we assign the simplices of K_i labels from index set $[m]$ in filtration order, relabeling K_{i+1} accordingly using ρ_i. As p is now a strictly increasing sequence, we may

compute the longest increasing subsequence $\mathrm{LIS}(q)$ and use this subsequence to recover a longest common subsequence (LCS) between (K_i, K_{i+1}), which we denote later with $\mathrm{LCS}(K_i, K_{i+1})$. We pass q and $\mathrm{LIS}(q)$ to our greedy scheduling algorithm, which returns as output an ordered set of move permutations $\mathcal{S}$ of minimum size, which we call a *schedule* (see Definition 1). Note one need not explicitly store the entire family of filtrations nor the schedules between them—Algorithm 1 may be easily modified to be completely *online*, keeping at most two filtrations and one decomposition in memory at any given time.

Algorithm 1 Scheduling algorithm

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

procedure MoveSchedule($\mathcal{K} = (K_1, K_2, \ldots, K_n)$)
  $\mathcal{S} = \emptyset$
  for $i = 1$ **to** $n - 1$ **do**
    $(p, q) \leftarrow ([m], \mathrm{Im}\, f_i^*)$
    $\mathrm{lis}_q \leftarrow \mathrm{LIS}(q)$ ▷ $O(m \log\log m)$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathrm{GreedySchedule}(q, \mathrm{lis}_q)$ ▷ $O(d^2 \log m)$
  $(R, V) \leftarrow$ Reduction($D = \partial K_1$)
  for (i, j) **in** $\mathcal{S}$ **do**
    $(R, V) \leftarrow$ **if** $i < j$ MoveRight(i, j) **else** MoveLeft(i, j)

In the following subsections, we investigate whether we may profitably exploit the increased flexibility Busaryevs move framework yields us. It was hypothesized that the reason the vineyards algorithm was more expensive than the naïve approach is due to the extra overhead of maintaining the decomposition at each transposition. Thus, in theory, decreasing the number of times the decomposition is restored to a valid state ought to reduce the total number of column operations needed to update a decomposition. This motivates the question: can one simultaneously minimize the number of times the decomposition is restored to a valid state and retain an efficient update scheme for *arbitrary* permutations?

3.1 Continuous setting

In the continuous time case, we are given a homotopy $F : K \times [0, 1] \to \mathbb{R}$ interpolating between a pair of filtrations $(K, f), (K, f')$ satisfying $f = F(\cdot, 0)$ and $f' = F(\cdot, 1)$, respectively. As the choice of homotopy F determines the number of transpositions that transform f into f' via F, bounding this number requires assumptions on F. If we assume the curves $\tau \mapsto F(\cdot, \tau)$ are in general position in $[0, 1] \times \mathbb{R}$ and that each pair of curves cross at most once, then the functions $\tau \mapsto F(\cdot, \tau)$ belong to a class of x-monotone curves called *pseudo-segments*. This family includes the straight-line homotopy $F(\sigma, \tau) = (1-\tau)f(\sigma)+\tau f'(\sigma)$, studied in the original vineyards paper [2]. Detecting all k intersections of m

pseudo-segments is a well-studied problem in computational geometry that can be optimally solved in output-sensitive $O(m \log m + k)$ time by several algorithms [26], where k is the output-sensitive term.

Given two filtrations $(K, f), (K, f')$, observe their simplexwise representations induced by f and f' can be thought of as elements of the symmetric group S_m. As applying permutations to K implies additional column operations on the underlying matrix decomposition, we think of any set of such permutations acting on the initial decomposition $R = DV$ of (K, f) as a *schedule* with respect to the pair f, f'. We formalize this with a definition.

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:*

$$R = D_0 V_0 \xrightarrow{s_1} D_1 V_1 \xrightarrow{s_2} \ldots \xrightarrow{s_d} D_d V_d = R' \tag{6}$$

where, for each $i \in [d]$, $R_i = D_i V_i$ is a valid decomposition respecting invariants 2.1, and R' is a valid decomposition of K'.

Let $F : K \times [0, 1] \to \mathbb{R}$ denote a homotopy of x-monotone curves. Then F decomposes into an ordered set of adjacent transpositions S_F given by pairs of simplices in K that change in their relative ordering between K and K':

$$\begin{aligned} S_F &= \{(i, i+1) \mid f(\sigma_i) < f(\sigma_{i+1}),\ f'(\sigma_i) > f'(\sigma_{i+1}),\ i \in [m]\} \\ &= \{s_1, s_2, \ldots, s_k\} \end{aligned}$$

Moreover, let $p = [m]$ and let $q = \operatorname{Im} \rho^*$, where $\rho^* : [m] \to [m]$ is the one-to-one correspondence induced by the reindexing bijection $\rho : K \to K'$. Then S_F is given by the inversions between p and q:

$$S_F = \operatorname{Inv}(p, q) = \{\, (i, i+1) \mid p(i) < p(i+1),\ q(i) > q(i+1),\ i \in [m] \,\}$$

The cardinality of S_F is exactly the *Kendall-τ* distance [27] between p and q:

$$K_\tau(p, q) = |\operatorname{Inv}(p, q)| \tag{7}$$

This implies the $|S_F| \sim O(m^2)$ in the worst case, achieved when $f = -f'$, which the grayscale image data example from section 1.3 exhibits. As the 9x9 patch contains $(81, 208, 128)$ simplices of dimensions $(0, 1, 2)$, the largest number of transpositions a schedule simplices in total. As the homotopy is simulated, observe $\approx$ 70,000 transpositions are generated, approaching the upper bound of , due to the fact that f' is nearly the reverse of f.

If our goal is to decrease the size of $|S_F|$, one option is to *coarsen* S_F to a new schedule $\widetilde{S}_F$ by collapsing contiguous sequences of adjacent transpositions

to moves, via the map:

$$(i, i+1)(i+1, i+2)\cdots(j-1, j) \mapsto (j, i+1, \cdots, j-1, i) \quad \text{if } i < j \tag{8}$$

We expect $\widetilde{S}_F$ to be cheaper to execute than S_F, with the tradeoff that less diagrams are produced. We make this precise next:

Proposition 2. *Let K denote a filtration of size $|K| = m$ with decomposition $R = DV$, T the maximum number of $O(m)$ operations required to execute S_F via vineyards, and M be the maximum number of $O(m)$ operations required to execute $\widetilde{S}_F$ with moves. Then, the inequality $M \leq \frac{1}{2}T$ holds.*

Proof First, consider executing the vineyards algorithm on a given schedule S_F. 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 3. 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 the assignment to and from d_* (lines (4), and (7) in RestoreRight of MoveRight in Algorithm 3) can be completed in $O(1)$ 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 swap their column pointers in $O(1)$ before proceeding with the necessary column operations. After the movement, d_* contains the newly modified column and $\text{col}_*(k)$ contains the unmodified donor d'_*, so we swap them once more in $O(1)$ time. Since we require at most one $O(m)$ column operation 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. □

In terms of size bounds, though clearly $|\widetilde{S}_F| \leq |S_F|$ and the associated coarsened $\widetilde{S}_F$ requires just $O(m)$ time to compute, the coarsening depends entirely on the initial choice of F—thus the quadratic upper bound remains, as its always possible that there are no contiguous subsequences to collapse. As mentioned in 2.2, this quadratic scaling induces a number of issues in the practical implementations of the vineyards algorithm.

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 into its position in the filtration given by K' in the order given by the latter—which we call *naive strategy*. However, it's not immediately clear whether this bound

is tight. In this section, we investigate the tightness of this bound by revisiting the problem 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:

$$s_d \circ \cdots \circ s_2 \circ s_1 \circ p = q \tag{9}$$

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, a 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 affects 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:

$$\begin{pmatrix} 1 \cdots i-1 & \overline{i \quad i+1 \quad \cdots \quad j-1 \; j} & j+1 \cdots m \\ 1 \cdots i-1 & \overline{i+1 \quad \cdots \quad j-1 \quad j \quad i} & j+1 \cdots m \end{pmatrix} \tag{10}$$

In cycle notation, this corresponds to the cyclic permutation:

$$(\; i \;\; j \;\; j\text{-}1 \;\; \ldots \;\; i\text{+}2 \;\; i\text{+}1 \;) \tag{11}$$

Similarly, in the context of edit operations, 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]:

$$\mathrm{d}_{\mathrm{lcs}}(p, q) = m + n - 2|\mathrm{LCS}(p, q)| \tag{12}$$

In general, one can compute the LCS itself in $O(mn)$ easily with dynamic programming. One might hope computing the size $\mathrm{d}_{\mathrm{lcs}}(p, q)$ alone exhibits a lower complexity, however there is substantial evidence that for general string inputs the complexity cannot be much lower than quadratic [30]. However, if the pair of filtrations K, K' are from the same underlying complex S, then they may both be thought of a permutations in S_m—the corresponding edit distance d then reduces to the *permutation edit distance* problem. 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'. Moreover, $|S^*| = d$ can be determined in $O(m \log\log m)$ time.

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 ι has with p' must also be strictly increasing. Thus, the problem of computing $\mathrm{LCS}(p, q)$ reduces to the problem of computing the *longest increasing subsequence* (LIS) of p', which can done in $O(m \log\log m)$ time using van Emde Boas trees [31]. The optimality of d follows from the optimality of the well-studied LCS problem [32]. □

Establishing a connection between the permutation edit distance and move scheduling allows use to exploit the combinatorial structure that comes from the developed theory on both LCS's and LIS's, which are both well-studied objects. We record a single corollary to demonstrate this fact.

Corollary 1. *If $(K, f), (K, f')$ are random filtrations of a common complex K of size m, then the size of longest common subsequence of simplices between f, f' is no larger than $m - \sqrt{m}$ in expectation, with probability 1 as $m \to \infty$.*

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

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

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 1. *Note the quantity from Corollary 1 captures the size of S^* between pairs of uniformly sampled permutations, as opposed to uniformly sampled filtrations, which have more structure. 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 1.3 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 simulation of persistence dynamically.

Theorem 1. *Given a pair of filtrations $(K, f), (K, f')$ and a decomposition $R = DV$ of K, the size of a minimal sequence $\mathcal{S} = (s_1, s_2, \ldots, s_d)$ of cyclic 'move' permutations $s_k = (i_k, j_k)$ satisfying:*

$$R = D_0 V_0 \overset{s_1}{\to} D_1 V_1 \overset{s_2}{\to} \ldots \overset{s_d}{\to} D_d V_d = R' \tag{13}$$

can be determined in $O(m \log\log m)$ time and $O(m)$ space, where $R' = D_d V_d$ denotes a valid decomposition of (K, f'). Moreover, if $i_k < j_k$ for all $k \in [d]$, then computing (13) requires $O(d\kappa)$ column operations, where:

$$\kappa = \sum_{k=1}^{d} (|\mathbb{I}_k| + |\mathbb{J}_k|)$$

where the quantities $|\mathbb{I}_k|, |\mathbb{J}_k|$ of the intermediate R_k, V_k are given in Proposition 1. Note that since κ depends on the sparsity of the intermediate entries $V_1, V_2, \ldots, V_d$ and $R_1, R_2, \ldots, R_d$, the bound $O(d\kappa)$ is output-sensitive.

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 κ follows directly from Algorithm 3. □

3.3 Constructing schedules

It is clear from Corollary 1 that one may compute the LCS between two permutations $p, q \in S_m$ via the LIS of a single permutation p^*, and that computation may be carried out in $O(m \log\log m)$ time. It is not immediately clear, however, how to obtain a sorting $p \mapsto q$ from a given $\mathcal{L} = \mathrm{LIS}(p')$ 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.

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 of symbols in $\mathcal{L}$ common to both p and q satisfies:

$$p^{-1}(\sigma) < p^{-1}(\tau) \implies q^{-1}(\sigma) < q^{-1}(\tau) \quad \forall \sigma, \tau \in \mathcal{L} \tag{14}$$

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 to symbols 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:

$$|\mathrm{LCS}(s \circ p, q)| = |\mathrm{LCS}(p, q)| + 1 \tag{15}$$

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

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

$$p^{-1}(\mathcal{T}_{\mathrm{pred}}(\sigma)) < p^{-1}(\sigma) < p^{-1}(\mathcal{T}_{\mathrm{succ}}(\sigma)) \tag{16}$$

for every $\sigma \in \mathcal{T}$. Now, suppose we choose a symbol $\sigma \notin \mathcal{T}$. If $p^{-1}(\sigma) < p^{-1}(\mathcal{T}_{\mathrm{pred}}(\sigma))$, then we must move σ to the right in p such that (16) holds. Similarly, if $p^{-1}(\mathcal{T}_{\mathrm{succ}}(\sigma)) < p^{-1}(\sigma)$, then we must move σ left in p to increase the size of the LCS. Assuming the structure $\mathcal{T}$ 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, which for completeness is shown via Algorithm 5 in the appendix.

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:

$$\mathrm{cost}(\mathcal{S}) = \sum_{k=1}^{d} |\mathbb{I}_k| + |\mathbb{J}_k| \tag{17}$$

where $|\mathbb{I}| + |\mathbb{J}|$ are the quantities from Proposition 1. One advantages to 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—no more than $O(m)$ time with row-oriented sparse matrices. It is natural to consider whether a greedy-type solution which chooses the minimal cost choice at every step yields an efficient strategy. We give a counter-example below demonstrating that a greedy procedure may lead to arbitrarily bad behavior.

Counter-Example: A pair of filtrations is given below, each 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 given below:

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

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

The subset of the filtration which corresponds to the simplices which lie in the LCS between these permutations is colored in red. For this example, the edit distance is $d = m - |\mathrm{LCS}(f, f')|$, implying 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 symbol x to the position of symbol y. The cost of each move operation and each schedule is recorded in Table 1. Note the greedy strategy which always selects the

Table 1: Move schedule costs

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

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 depending on the tie-breaker. While the cheapest schedule S_1 is in this candidate set, so is S_6, the most expensive schedule. As a result, a greedy strategy which chooses the lowest-cost move may not yield an optimal schedule.

3.4.2 Proxy objective

Although the greedy approach from the last section can lead to arbitrarily high cost schedules, we find similar greedy-like strategies can yield computationally efficient heuristics in practice, even if they are suboptimal. We seek a fast procedure for generating schedules that is not only substantially better than a random or simple schedule strategy in term of column reductions, but has a low enough time and storage complexity to be practical for larger data sets.

We seek a *proxy objective* that correlates with (17) and does not explicitly depend on the entries in the decomposition, as minimizing these terms directly is difficult due to the changing sparsity of the intermediate matrices R_k, V_k, Given a pair of filtrations $(K, f), (K, f')$, consider a schedule $\mathcal{S} \in \Phi(f, f', d)$ of cyclic permutations $((i_1, j_1), (i_2, j_2), \ldots, (i_d, j_d))$ minimizing:

$$\widetilde{\text{cost}}(\mathcal{S}) = \sum_{k=1}^{d} 2|i_k - j_k| \geq \sum_{k=1}^{d} (|\mathbb{I}_k| + |\mathbb{J}_k|) \tag{18}$$

In practice, this bound is very loose due to the sparsity of both R and V. Nonetheless, the complexities of the move operations discussed in Section 2.3 depend on $|i - j|$, and minimizing (18) has the intuitive interpretation as minimizing *net displacement*. A similar ℓ_1-type distance for measuring the disarrangement between permutations is the *Spearman distance*, defined as:

$$F(p, q) = \sum_{i=1}^{m} |p(i) - q(i)| = \sum_{i=1}^{m} |i - (q^{-1} \circ p)(i)| \tag{19}$$

The Spearman distance shares any similarities with K_τ: it is a metric on S_m that is invariant under consistent relabeling. Indeed, Diaconis et al. [27] showed the Spearman distance approximates K_τ within a factor of two:

$$K_\tau(p, q) \leq F(p, q) \leq 2K_\tau(p, q) \tag{20}$$

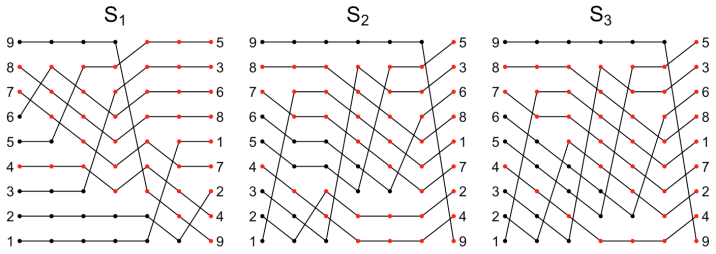
Moreover, in contrast to K_τ, the Spearman distance can be computed in $O(m)$ time, has a ℓ_1 interpretation between ranking, and is often used in the well-known *rank aggregation problem*: whereas obtaining a Kemeni optimal aggregation with respect to K_τ is NP-hard, the optimal such aggregation with respect to F is obtainable in polynomial time [35]. To adapt $F(\cdot, \cdot)$ to sortings, we decompose the footrule distance additively via the bound:

$$\hat{F}_{\mathcal{S}}(p, \iota) = \sum_{i=1}^{d-1} F(\hat{s}_i \circ p, \hat{s}_{i+1} \circ p) \geq F(p, \iota) \tag{21}$$

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 $p \mapsto \iota$. The problem of minimizing the right hand side of (21) can be interpreted as a crossing minimization problem

for a set of k-layered 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, observe that $F(p, m_{i,j} \circ p)$ is twice the number of edge crossings in the graph, and that equality in (21) is achieved when the displacement of each symbol 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 derived in section 3. Unfortunately, the k-layer crossing minimization problem is NP-hard for k sets of permutations, when $k \geq 4$ [36].

Example: 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 how each symbol has been moved. Black/red vertices correspond to symbols in and outside of 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 $K_\tau(p, q) = 21$.

In light of the discussion above, we propose a heuristic strategy to minimize (21) which we observed is both efficient to compute and effective in practice. The heuristic is inspired by the simplicity of computing the Spearman distance between cyclic permutations at the schedule construction phase. Suppose we begin with 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. Since the Spearman distance is simply the sum of the absolute value of these displacements, at any point during during the execution of Algorithm 5 we may obtain $F(\bar{p}, q)$ simply by having access to the sum of every entry in $\mathcal{A}$. There are two degrees of freedom in Algorithm 5: the first is in the choice of $\sigma \in \mathcal{D}$, and the second is in choosing j. If we fix a heuristic for the latter, then we have a set of possible valid moves $S_{\mathcal{D}}$ induced by each $\sigma \in \mathcal{D}$ to choose from. Observe that each permutation $s_\sigma \in S_{\mathcal{D}}$

changes the displacement of every symbol in at most three different ways:

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

Thus, if we replace $\mathcal{A}$ with a data structure supporting $O(\log m)$ access time to aggregate information and $O(\log m)$ modification time ability on $|i-j|$ entries simultaneously, we could greedily choose the next permutation s to minimize:

$$s_{\text{greedy}} = \underset{\text{valid } s_\sigma \in S_{\mathcal{D}}}{\arg\min} \; F(s \circ p, \iota) \tag{22}$$

in $O(d \log m)$ time. The former problem reduces to the problem of efficiently calculating *prefix sums*, which is easily solved. It is not immediately clear how to achieve the latter modification complexity in $O(\log m)$ time, since $|i-j| \leq m$ is potentially larger than $O(\log m)$—to handle this, note that one can apply constant-factor updates to multiple values in a *implicit treap* data structure in $O(\log m)$ time via *range updates*. Since single element modifications, removals, and insertions can all be achieved in $O(\log m)$ expected time with such a data structure, we conclude that the greedy approach solving equation (22) requires just $O(d^2 \log m)$ time.

4 Applications and Experiments

4.1 Video data

A common application of persistence is to 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 using minimal sized scheduled and the scalability of the greedy approach proposed in section 3.4.2, we re-use the video data from 1.3 as a baseline benchmark. We perform two performance tests: one to one to test the impact of shrinking the number of permutations a given reindexing operation needs and one to test the asymptotic behavior of the greedy approach and.

In the first test, we fix a grid size of 9×9 and record the cumulative number of column operations needed to simulate persistence dynamically across 25 evenly-spaced time points using a variety of scheduling strategies. The primary 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. In particular, if $d = m - |\text{LCS}(f, f')|$ we use a parameter $\alpha \in [0, 1]$ to choose $m - \alpha \cdot d$ random symbols to move using the same construction method outlined in section 3.3. When $\alpha = 0$, the strategy reduces to using selection sort to reindex $f \mapsto f'$ using a random ordering of simplices; otherwise, $\alpha = 1$ reduces to using a

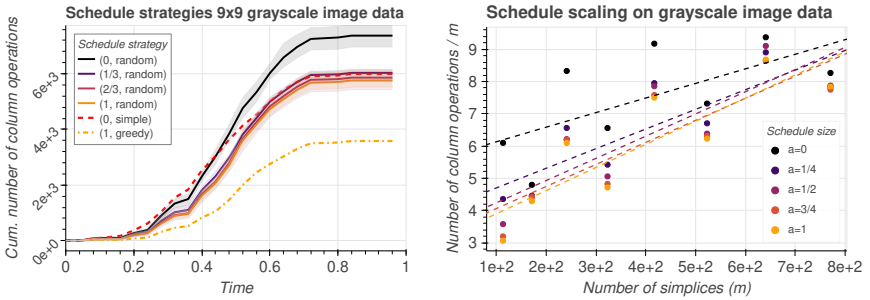


Figure 2: Performance data

minimal sized schedule (with a random ordering of simplices). The results are summarized in the left graph on Figure 2. To get an idea of the variation of the random sampling, we run 10 independent iterations and shade the upper and lower bounds of the schedule costs, which is shown around the mean schedule cost depicted by the solid lines in Figure 2. As one can see from the figure, while using less move operations (lower α) does progressively 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 a 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 1.3, each of increasing grid sizes of 5×5, 6×6, $\ldots$, 12×12. For each data set, we simulate 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 2. On the vertical axis, we plot the total number of column operations needed to simulate the video again across 25 evenly-spaced time points *as a ratio* of the data set size (m); we also show the regression curves one obtains from considering each schedule size. 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, using schedules with minimal size tended to be 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

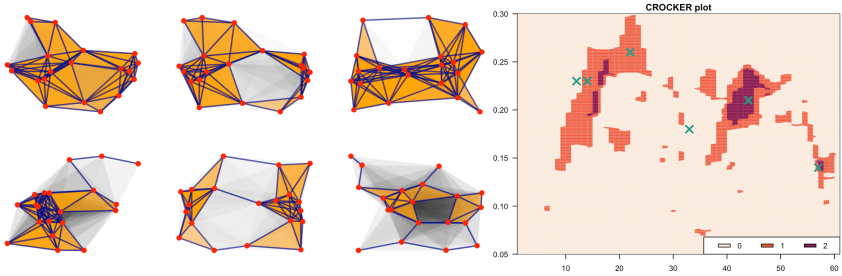


Figure 3: A crocker plot (right) depicts the evolution of dimension $p = 1$ Betti curves over time. The green X marks correspond chronologically to the complexes (left), in row-major order. The large orange and purple areas depict 1-cycles persisting in both space (y-axis) and time (x-axis).

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*[4] 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 3. 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 1.2. 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

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

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:*

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

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.

We test the efficiency of computing the necessary information to generate these crocker stacks using a spatiotemporal data set to illustrate the applicability of our method. Specifically, we ran a *flocking* simulation similar to the simulation run in [24] with $m = 20$ vertices moving around on the unit square equipped with periodic boundary conditions (i.e. $S^1 \times S^1$). We simulated movement by equipping the vertices with a simple set of rules which control how the individual vertices position change over time. Such simulations are also called *boid* simulations, and they have been extensively used as models to describe how the evolution of collective behavior over time can be described by simple sets of rules. The simulation is initialized with every vertex positioned randomly in the space; the positions of vertices over time is 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 of the simulations. As depicted in Figure 4, our move strategy is far more efficient than both vineyards and the naive `pHcol` strategies.

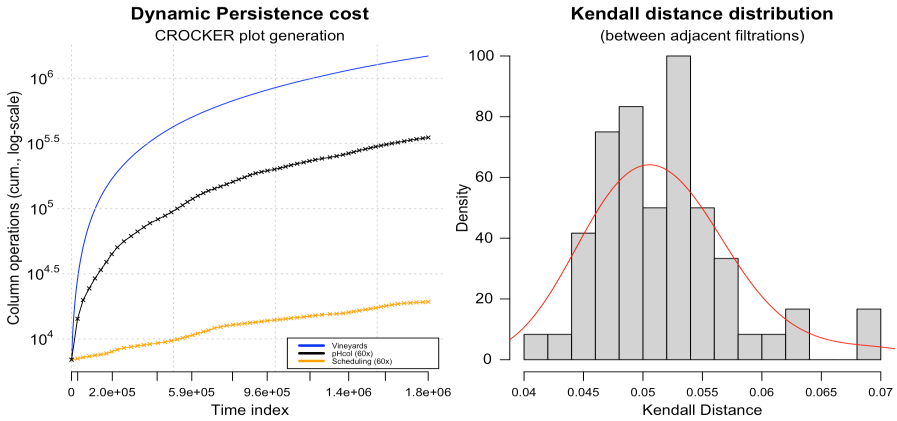


Figure 4: One the left, the cumulative number of column operations (log-scale) of the three baseline approaches tested. On the right, the normalized K_τ between adjacent filtrations depicts the coarseness of the discretization—about 5% of the $\approx O(m^2)$ simplex pairs between adjacent filtrations 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 [37]. Indeed, one of the drawbacks of persistence is its instability with respect to strong outliers, which can obscure the detection of significant topological structures [38]. One exemplary use case of multi-parameter persistence is to detect these strong outliers by filtering the data with respect to both density and the associated metric. 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 exploratory 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 supplementary details for the curious reader in the appendix A.2.

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 5. 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 outlined 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. 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 worse-case complexity of the barcode templates computation $O(m^5)$—this is both the highest complexity and

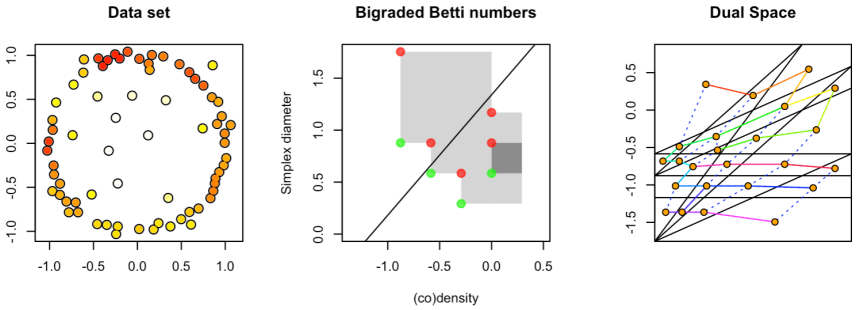


Figure 5: Bipersistence example on an 8×8 coarsened grid. On left, the input data, colored by density. In the middle, the bigraded Betti numbers $\beta_0(M)$ and $\beta_1(M)$ (green and red, respectively), the dimension function (gray), and a line L emphasizing the persistence of features with high density. On the right, the line arrangement $\mathcal{A}(M)$ lying in the dual space derived from the $\beta(M)$.

most time-intensive sub-procedure the RIVET software [11] depends on. Despite this significant complexity barrier, there is room for optimism: in practice, the external stability result from [39] justifies the use of a coarsening procedure which approximates the module M with a smaller module M' via a grid-like reduction, 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 1 was designed for precisely such a computation, 2-parameter persistence is prototypical of the class of methods that stand to benefit from move scheduling.

4.3.2 Natural images dataset

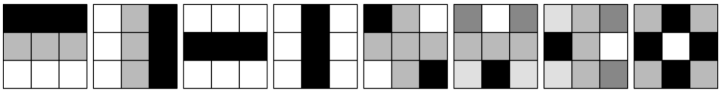
A common hypothesis is that high dimensional data tend to lie in the vicinity of an embedded, low dimensional manifold or topological space. An exemplary demonstration of this is given in the analysis by Lee et al. [40], who explored the space of high-contrast patches extracted from Hans van Hateren's [41] still image collection[5], which consists of $\approx 4,000$ monochrome images depicting various areas outside Groningen (Holland). In particular, [40] 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

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

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 $x^T Dx$ 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) basis, they 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.

After Lee et al published their work, Carlsson et al. [42] subsequently performed extensive experiments using persistent homology, wherein he found 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 $\mathcal{M}$ which accounts for only 21% of the 7-spheres volume. Along a similar vein in the sparse coding context, Perea et al. [43] introduced a dictionary learning framework for estimating the distribution of patches from texture images.

If one was not aware of the analysis done by [40–43], 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 reveal the underlying the topological space with persistent homology [42]. 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.

Consider the (coarsened) fibered barcode computed from a standard Rips / codensity bifiltration on a representative sample of the image data from [41], shown in Figure 6. 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 6), wherein the first Betti number is 5. Further inspection suggests one plausible candidate is the three-circle

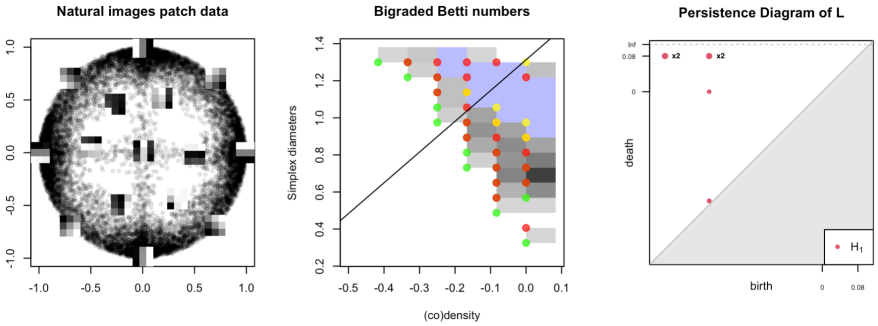


Figure 6: Bipersistence example of natural images data set on an 12×16 coarsened grid. On the left, a projection the full data set is shown, along with the 15 landmark patches. (Middle) the bigraded Betti numbers and a fixed line L over parameter space. As before, the 0/1/2 dimension bigraded Betti numbers 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.

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 of leads to the projection shown in the top left of Figure 6, 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 [42] 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

As we have outlined the computational theory of 2-parameter persistence and elucidated its relevance to our proposed move scheduling approach, we now demonstrate the efficiency of scheduling using the same high-contrast patch data set studied in [40] by evaluating the performance of various methods at computing the fibered barcode invariant via the parameterization from A.2.

Due to the aforementioned high complexity of the fibered barcode computation, we begin by working with a subset of the image patch data $\mathcal{X}$. In particular, 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 2. We also record the number of 2-cells in $\mathcal{A}(M)$ and the number of permutations applied throughout the 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 2: 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, when there is a high degree of coarsening, vineyards is particularly inefficient and moves requires only about 3x less column operations that naively computing $\mathcal{T}$ independently. As the coarsening becomes more refined and more 2-cells are added to $\mathcal{A}(M)$, however, vineyards quickly becomes a much more viable option compared to `pHcol`—as predicted—though even at the highest coarsening we tested the gain in efficiency is relatively small. In contrast, our proposed moves approach scales quite well with this refinement, requiring about 12% and $\approx 5\%$ of the number of column operations as vineyards and pHcol, respectively.

5 Conclusion and Future Work

In conclusion, we presented a scheduling algorithm for efficiently updating a decomposition in 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. As mentioned in section 1.2, examples include:

1 Accelerating PH featurization methods for time-varying systems
2 Optimization procedures involving persistence diagrams
3 Detecting homological critical points in time-varying filtrations

Indeed, we see our approach as potentially useful to any situation where the structure of interest is 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 is "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. 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[6], permuting both the rows and columns generally takes at most $O(Z)$ 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 (17) or (21) to a better-known problem.

[6] 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)

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] 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

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

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

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

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

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

[38] 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

[39] Landi, C.: The rank invariant stability via interleavings. arXiv preprint

arXiv:1412.3374 (2014)

[40] 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)

[41] 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)

[42] 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)

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

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

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

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

[47] 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 2. The algorithm begins by copying D to a new matrix R, to be

Algorithm 2 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 2, 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 4. Moreover, since R is reduced in all but one column by line 6 in Algorithm 4, we can still guarantee the number of column operations in R will scale with $|i-j|$. For a supplementary description of the move algorithm, see [12].

Algorithm 3 Move Right Algorithm

```
1: function RestoreRight(R, V, 𝕀 = {I_1, I_2, ..., I_s})
2:     (d_low, d_R, d_V) ← (low_R(I_1), col_R(I_1), col_V(I_1))
3:     for k in I_2, ..., I_s do
4:         (d'_low, d'_R, d'_V) ← (low_R(k), col_R(k), col_V(k))
5:         (col_R(k), col_V(k)) += (d_R, d_V)
6:         if d'_low < d_low then
7:             (d_low, d_R, d_V) ← (d'_low, d'_R, d'_V)
8:     return (R, V, d_R, d_V)
1: function MoveRight(R, V, i, j)
2:     𝕀 = columns satisfying V[i, i : j] ≠ 0
3:     𝕁 = columns satisfying low_R ∈ [i : j] and row_R(i) ≠ 0
4:     (R, V, d_R, d_V) ← RestoreRight(R, V, 𝕀)
5:     (R, V) ← RestoreRight(R, V, 𝕁)
6:     (R, V) ← (PRP^T, PVP^T)
7:     (col_R(j), col_V(j)) ← (Pd_R, Pd_V)
8:     return (R, V)
```

We recall an important claim given in [12] on the effect that move operations have on the status of simplices in the pairing. Recall from section 2 that simplices which create new homology classes are called *creators* and simplices that destroy homology classes are called *destroyers*. The effect of the movement on intermediate simplices depends on the direction of the movement. If $i < j$ (respectively, $j < i$), all simplices at positions $k \in [i+1 \ : \ j]$ are shifted down (respectively, up) by 1.

Algorithm 4 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 $l < r$, $\text{low}_R(l) = \text{low}_R(r)$ maximal
  while $\text{low}_R(l) \neq 0$ **and** $\text{low}_R(r) \neq 0$ **do**
    $(\text{col}_R(r), \text{col}_V(r)) \mathrel{+}= (\text{col}_R(l), \text{col}_V(l))$
    $\mathbb{K} \leftarrow \mathbb{K} \setminus l$
    $(l, r) \leftarrow$ indices $l, r \in \mathbb{K}$ satisfying $l < r$, $\text{low}_R(l) = \text{low}_R(r)$ maximal
  return (R, V)

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

A.1.2 LCS-Sort

Here we record explicitly the simple 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 (15), but nonetheless for posterity sake we record it here for the curious reader; it 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 $\bar{p} \mapsto p$ via operations which (strictly) increase the size of $\mathcal{L}$, we ensure that the size of the set of corresponding permutations is exactly $m - |\mathcal{L}|$.

Suppose $\mathcal{L}$ has been computed from $\bar{p}$. Since $\mathcal{L}$ is strictly increasing, the only symbols left to permute are in $\mathcal{L} \setminus \bar{p}$, which we denote with $\mathcal{D}$. After choosing any symbol $\sigma \in \mathcal{D}$, 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 maintain a data structure $\mathcal{T}$ which enables us to query the successor and predecessor of any given symbol $s \in \bar{p}$ *in* $\mathcal{L}$. We also require a data structure to query the position of a given element $\sigma \in \bar{p}$, which for now we simply use the inverse permutation $\bar{p}$ (though a more efficient representation can be used based off of symbol *displacements*, see section 3.4.2. After the symbol is

Algorithm 5 Sorting algorithm

function LCS-SORT(p, q)
  $\bar{p} \leftarrow q^{-1} \circ p$
  $\mathcal{L} \leftarrow \mathrm{LCS}(p, q) = \mathrm{LIS}(\bar{p})$ ▷ $O(m \log\log m)$
  $(\mathcal{S}, \mathcal{D}, \mathcal{T}) \leftarrow (\emptyset, \bar{p} \setminus \mathcal{L}, \mathcal{L})$
  while $\mathcal{D}$ is not empty **do**
    $\sigma \leftarrow$ arbitrary element in $\mathcal{D}$
    $(i, i_p, i_n) \leftarrow (\bar{p}^{-1}(\sigma), \bar{p}^{-1}(\mathcal{T}_{\mathrm{pred}}(\sigma)), \bar{p}^{-1}(\mathcal{T}_{\mathrm{succ}}(\sigma)))$ ▷ $O(\log\log m)$
    if $i < i_p$ **then**
      $j \leftarrow$ arbitrary element in $[i_p, i_n)$
    else $i_n < i$
      $j \leftarrow$ arbitrary element in $(i_p, i_n]$
    $(\mathcal{S}, \mathcal{D}, \mathcal{T}) \leftarrow (\mathcal{S} \cup (i, j), \mathcal{D} \setminus \sigma, \mathcal{T} \cup \sigma)$ ▷ $O(\log\log m)$
    $\bar{p}^{-1} \leftarrow \bar{p}^{-1} \circ m_{ij}^{-1}$ ▷ $O(m)$
  return $\mathcal{S}$

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 $p \mapsto q$ (or equivalently, $\bar{p} \to \iota$ are stored in an array $\mathcal{S}$, which is then returned for further use.

A.2 2-parameter persistence

We now describe the reparameterization between the bigraded Betti numbers and the set of "critical lines" Lesnick and Wright [37] 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:

$$\begin{aligned} \mathcal{D}_\ell : \mathcal{L} &\to [0,\infty) \times \mathbb{R} & \mathcal{D}_p : [0,\infty) \times \mathbb{R} &\to \mathcal{L} \\ y = ax + b &\mapsto (a, -b) & (c, d) &\mapsto y = cx - d \end{aligned} \tag{23}$$

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* $\mathrm{push}_L(a) : \mathbb{R}^2 \to L \cup \infty$ as:

$$\mathrm{push}_L(a) \mapsto \min\{v \in L \mid a \leq v\} \tag{24}$$

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

1 For $r < s \in \mathbb{R}^2$, $\mathrm{push}_L(r) \leq \mathrm{push}_L(s)$
2 For each $a \in \mathbb{R}^2$, $\mathrm{push}_L(a)$ 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, 44] 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:

$$\mathcal{B}_L(M) = \{[\,\text{push}_L(a),\, \text{push}_L(b)\,) \mid (a, b) \in \mathcal{T}^e, \text{push}_L(a) < \text{push}_L(b)\} \quad (25)$$

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.2.1 Invariant computation

Computationally, the algorithm from [45] 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 2 [45]. 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 [46]. 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 2 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 [47], and thus can be used so long as the distance function between templates is a valid metrics. [37] 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.