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

Keywords Computational topology, Persistent homology, Topological data analysis

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 in the size of the underlying filtration \[4\]—their adoption to dynamic settings poses a challenging computational problem. With the current state of the art tools, there is no recourse to the problem of computing persistence from a time-varying filtration containing millions of simplices at thousands of snapshots in time. Nonetheless, having the capability of efficiently

---

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

†Khoury College of Computer Sciences, Northeastern University.

‡Department of Mathematics and Khoury College of Computer Sciences, Northeastern University.

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.
updating the persistence of a time-varying filtration has far-reaching consequences: methods that vectorize persistence diagrams for machine learning purposes, like adaptive template functions \(^5\) and persistence images \(^6\), or summary statistics like \(\alpha\)-smoothed Betti curves \(^7\), all immediately become computationally viable tools in dynamic settings.

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 given a time-varying filtration. The vineyards approach can be interpreted as an extension of the `matrix reduction` algorithm \(^8\), which computes the persistence diagram of a fixed filtration \(K\) with \(m\) simplices in \(O(m^3)\) time. The main step in the reduction algorithm is a specific decomposition \(R = DV\) of the boundary matrix \(D\) of \(K\). Since \(V\) is always full rank and upper-triangular, an alternative is to compute a \(D = RU\) decomposition, where \(U = V^{-1}\). The vineyards algorithm transforms a time-varying filtration into an ordered set of permutations to be applied to adjacent rows and columns of the decomposition \(D = RU\), 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. This algorithm is also efficient asymptotically: if there are \(d\) time-points where the filtration order changes, then vineyards takes \(O(m^3 + md)\) time; one \(O(m^3)\)-time reduction upfront for the filtration at time \(t_0\) followed by one \(O(m)\) operation to update the decomposition for the remaining time points \((t_1, t_2, \ldots, t_d)\). When \(d >> m\), this \(O(md)\) approach is far more efficient than the \(O(dm^3)\) “naive” strategy of computing the diagrams independently at every time point.

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 \(^1\). The reason for this disparity is 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, whether performing the reductions naively or updating the decomposition dynamically using vineyards, they observed maintaining the decomposition is the most computationally demanding aspect of RIVET \(^11\).

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 `continuous` time families of diagrams, it is not necessarily efficient when the time parameter is coarsely discretized. Our methodology is based on the observation that practitioners often don’t need (or want!) all \(d\) persistence diagrams generated by a continuous 1-parameter of filtrations; usually just \(n << d\) of them suffice. By exploiting the “donor” concept introduced in 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, dramatically 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 \(^3\), 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 available online as an open source \(R\) package\(^6\).

### 1.2 A Motivating Example and Contributions

As a motivating example, we describe a simple experiment illustrating why the vineyards algorithm does not always yield an efficient strategy for time-varying applications. Consider a series of grayscale images (i.e., a video) depicting a fixed-width annulus expanding about the center of a \(9 \times 9\) grid. Each image contains the same number of pixels, whose intensities vary with time. For each image, we build a filtered simplicial complex using the Freudenthal triangulation of the plane, with the order of inclusions obtained by lower stars and pixel values. Snapshots of the image data and its corresponding time-varying filtration is depicted in Figure \(^1\) (top right and bottom right, respectively), which we use as a baseline for comparing vineyards and the standard reduction algorithm phtco1 (Algorithm \(^1\)). Note that the size of the filtration is fixed—no simplices are ever added or removed over time. There are two events which dramatically alter the persistence diagrams extracted from these filtrations: the first occurs when the central connected component splits to form a cycle, and the second when the annulus splits into four components. As a consequence, in this example, only a few persistence diagrams are needed to capture the major topological changes over time.

---

\(^5\)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.

\(^6\)Code is available here: [https://github.com/peekzc/dart](https://github.com/peekzc/dart)
Figure 1: On the right, the grayscale image data set and corresponding lower star filtrations. On the left, the cumulative number of column operations required to compute persistence for this time-varying filtration. The Betti numbers for each of the shown steps in the filtration, from left to right, are: \((\beta_0, \beta_1, \beta_2) = (1, 0, 0), (1, 1, 0), (1, 1, 0), (4, 0, 0)\). Observe that computing the diagrams independently at 10 evenly spaced time points (the green line) both captures these major topological changes and is the most computationally efficient approach shown.

Consider the situation where one is concerned with capturing these dramatic changes in homology. On the left side of Figure 1, we compare the cumulative cost (in total number of column operations) of various approaches. Since it is unknown a priori at which times the persistent homology will change, one solution is to discretize the time domain at \(n\) evenly spaced points and compute the persistence diagrams at each one of them independently. An alternative approach is to construct a (e.g., straight-line) homotopy between contiguous time-points, and then decompose these homotopies into adjacent transpositions in the filtration order—the vineyards approach. The former is often used in practice, but it is only an approximation; the latter is guaranteed to capture all homological changes in persistence that occur in simulating the 1-parameter family. The graph in Figure 1 depicts two different size approximations (purple, \(n = 100\) and green, \(n = 10\)), wherein the reduction algorithm (\(pHcol\)) was applied at \(n\) time points, and two exact strategies (black and blue), which compute all \(\approx 7 \times 10^4\) diagrams given by the homotopy. As shown in the figure, if one needs to simulate the homotopy exactly, then 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 the vineyards strategy, while still capturing the main events.

The existence of a time discretization that is more efficient to compute than continually updating the decomposition, indicates that the vineyards algorithm must incur some overhead (in terms of column operations) to perform transpositions, even if the persistence diagrams remain unchanged. Even in situations where one is updating a decomposition to a filtration that is relatively "close"—e.g. any two adjacent filtrations in the bottom right of Figure 1—it is still more efficient to apply \(pHcol\) directly than to iteratively update the decomposition (see the case where \(n = 10\)). Moreover, any optimization to the reduction algorithm (e.g., clearing [13]) would only increase this disparity. See [14, 15, 10, 3] for relevant scenarios and applications.

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 bounds in expectation as well as efficient algorithms for achieving these bounds—both of which are derived from a reduction to the Longest Increasing Subsequence (LIS) problem. This reduction yields an efficient algorithm for generating sequences of moves \((s_1, s_2, \ldots, s_d)\) 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 persistent homology computations is given; our move schedules are demonstrated to be an order of magnitude more efficient than existing approaches in applications such as flock simulations and 2D persistence computations with RIVET.
1.3 Related Work

To the authors knowledge, work focused on ways of updating a decomposition \( R = DV \), for all homological dimensions, is limited—there is the vineyards algorithm [2] and the moves algorithm [12], both of which are discussed extensively in section 4.

In contrast, there is extensive work on improving the efficiency of computing a (static) \( R = DV \) decomposition. Chen [13] 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 [16], which dramatically improves the effectiveness of the clearing optimization [17]. There are many other optimizations on the implementation side: the use of ranking functions defined on the combinatorial number system enables implicit cofacet enumeration, removing the need to store the boundary matrix 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 \( \mathbb{Z} \) pivot entry retrieval, and so on [17, 18].

By making stronger assumptions on the underlying topological space, restricting the homological dimension, or targeting a weaker invariant (e.g. Betti numbers), one can usually obtain faster or alternative approaches recording information related to persistence. For example, Attali et al. [19] give a linear time algorithm (under the word RAM model of computation) for computing persistence on graphs. In the same paper, they describe how to obtain \( \epsilon \)-simplifications of 1-dimensional persistence diagrams for filtered 2-manifolds by using duality and symmetry theorems. Along a similar vein, Edelsbrunner et. al. [20] give a fast incremental algorithm for computing persistent Betti numbers up to dimension 2, again by utilizing symmetry, duality, and “time-reversal” [21]. Chen et. al. [22] give an output-sensitive method for computing persistent homology, utilizing the property that certain submatrices of \( D \) have the same rank as \( R \), which they exploit through fast sub-quadratic sparse-matrix rank algorithms.

If zeroth homology is the only dimension of interest, computing and updating both the persistence and rank information related to persistence. 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. [23] give a \( O(\epsilon) \)-per-update algorithm that allows one to maintain a merge tree with \( \epsilon \) edges, associated to the filtration parameter changing continuously over time. If only the zeroth-dimensional Betti numbers are needed for a particular application, this problem reduces even further to tracking the connected components of a dynamic graph—sometimes referred to as the dynamic connectivity problem, which can be efficiently solved in amortized \( O(\log n) \) query and update times using either Link-cut trees or multi-level Euler tour trees [24].

1.4 Outline

The remainder of 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. In particular, to the computation of crocker stacks from flock simulations, and to 2-dimensional persistence on the Mumford data set of image patches from natural images. In Section 5 we conclude the paper by discussing other possible applications and future work. We also include an appendix which introduces certain topics in more detail, such as the vineyards algorithm itself, for readers who may be unfamiliar but would like a more in-depth understanding.

2 Background

2.1 Notation

Suppose one has a family \( \{K_i\}_{i \in I} \) of simplicial complexes indexed by a totally ordered set \( I \), and so that for any \( i < j \in I \) we have \( K_i \subseteq K_j \). There are two index sets of interest in what follows: \( \mathbb{R} \) and \([n] = \{1, \ldots, n\} \). Such a family is called a filtration, which is deemed to be essential if \( i \neq j \) implies \( K_i \neq K_j \). Moreover, an essential filtration is said to be simplexwise if \( K_j \setminus K_i = \{\sigma_j\} \) whenever \( j \) is the immediate successor of \( i \) in \( I \). Any finite filtration may be trivially converted into an essential, simplexwise filtration via a set of condensing, refining, and reindexing
maps (see [17] for more details). As a result, and without loss of generality, here we exclusively consider essential simplewise filtrations. For brevity, we will simply refer to them as filtrations.

Let $K$ be an abstract simplicial complex and $F$ a field. A $p$-chain is a formal $F$-linear combination of $p$-simplices of $K$. The collection of $p$-chains under addition yields an $F$-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 $Z_p(K) = \text{Ker} \partial_p$. Similarly, the collection of $p$-boundaries forms $B_p(K) = \text{Im} \partial_{p+1}$. 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 $F$. 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:

$$H_p(K_1) \to H_p(K_2) \to \cdots \to H_p(K_m) \tag{1}$$

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 $\text{dgm}_p(K)$, 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.

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

$$\text{Rips}_\epsilon(X) = \{S \subseteq X : S \neq \emptyset \text{ and } \text{diam}(S) \leq \epsilon\} \tag{2}$$

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

Let $\mathbb{X}$ be a triangulable topological space. That is, so that there exists an abstract simplicial complex $\mathbb{X}$ 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.

Consider a homotopy $F(x, \tau) : \mathbb{X} \times [0, 1] \to \mathbb{R}$ and denote its “snapshot” at a given time-point $\tau$ 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 $\tau$ varies in $[0, 1]$, the points in $\text{dgm}_p(f_\tau)$ trace curves in $\mathbb{R}^3$, called vines and which together form a vineyard. The stability of persistence implies that these curves will be continuous if $F$ is continuous and the $f_\tau$’s are tame [1]. The vineyard analogy acts as a guidepost for practitioners seeking to understand how subtle changes occurring to the topological structure over time. A more detailed discussion of the vineyards approach is provided in section [2.3].

### 2.2 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 pfiltr [16]. The pseudocode is outlined in Algorithm [1]. Without optimizations, like clearing or implicit matrix reduction, the standard algorithm is very inefficient. Nonetheless, it serves as the foundation of most implementations that compute persistent homology, and its invariants are necessary before introducing both vineyards in section [2.3] and our move schedules in section [3].

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 $V$. Given a filtration $K$ with $m$ simplices and maximal dimension $d$, one assembles the elementary boundary chains $\partial(\sigma)$ as columns ordered according to the filtration, forming the filtration boundary matrix $D$. The reduction algorithm can compute the persistent homology for all dimensions up to $d - 1$. In this case, $D$ is a square matrix of size $m \times m$. Alternatively, a single dimension $i \leq d - 1$ can be computed, in which case one reduces a pair of sub-matrices $D_i$ and $D_{i+1}$ of $D$, where $D_i$ has dimension $d_{i-1} \times d_i$ and $D_{i+1}$ is $d_i \times d_{i+1}$. If one is interested in a particular dimension of homology, then the latter approach is preferred. However, the description of the algorithms is simpler when considering the full matrix, so $D$ will always be $(m \times m)$ in what follows. All algorithms discussed generalize to both situations. We give an example using the
Algorithm 1 Reduction Algorithm (PHc01)

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:            $(\text{col}_R(j), \text{col}_V(j)) \leftarrow (\lambda \cdot \text{col}_R(i), \lambda \cdot \text{col}_V(i))$
7:    return $(R,V)$

matrix 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, read as inclusions from left to right, is given as $K = (u,v,w,a,b,c)$. The reduction algorithm begins by setting $R = D$ and $V = I$, followed by left-to-right column operations until invariant I2 below is satisfied. If the $i$-th column of $R$ is nonzero, then $\text{low}_R(i)$ is the row index of its lowest nonzero entry. Using $\mathbb{Z}_2$ coefficients and homology in dimension 1 to simplify the presentation, the reduction proceeds as follows:

\[
R_1 = \begin{bmatrix} u & 1 & 1 \\ w & 1 & 1 \end{bmatrix}, \quad V_1 = \begin{bmatrix} a & 1 \\ b & 1 \\ c & 1 \end{bmatrix}
\]

Then

\[
R_1 = \begin{bmatrix} u & 1 & 1 \\ w & 1 & 1 \end{bmatrix}, \quad V_1 = \begin{bmatrix} a & 1 \\ b & 1 \\ c & 1 \end{bmatrix}
\]

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

It is clear from Algorithm 1 that a loose upper bound for this reduction is $O(m^3)$, where $m$ is the number of simplices of the filtration; it turns out that this bound is in fact tight, see [4] for more details. There are actually many variations and optimizations proposed over the past decade to Algorithm 1; for example, one can reverse the order of the filtration and compute cohomology [16] instead. Despite these variations in computation, regardless of how the decomposition is obtained, it must obey the following invariants:

**Decomposition Invariants:**

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

If these two invariants are satisfied, then we call the decomposition valid. Given any valid $R = DV$ decomposition, then [8]: The collection of persistent pairings is independent of the choice of $R$ and $V$. When reducing the boundary matrix, we can therefore perform column operations in any order, as long as columns are added from left to right, and so long as every column operation performed in $R$ is also performed on $V$. Once $R$ is reduced, then the low entries in $R$ yield the correct pairings encoding the persistence diagram of the corresponding filtration $K$.

2.3 Vineyards

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, and to detect homological critical events during the construction. Notice that homological critical events can only occur when the filtration order changes, though the converse is not true. Thus, detecting all such events reduces to computing matrix decompositions at a finite set of time points interleaved between changes in the filtration order. At the finest scale, these changes manifest as transpositions of adjacent simplices. As a result, a fixed set of rules for maintaining a valid $R = DV$ decomposition under adjacent transpositions is enough to simulate persistence dynamically. These rules prescribe certain column and row operations which must be
applied to the matrix decomposition either before, during, or after the transposition, in order to ensure decomposition invariants I1 and I2 are respected.

Let $S^j_i$ represent the upper-triangular matrix such that multiplying by it on the right is equivalent to adding column $i$ to column $j$. Multiplication by $S^j_i$ on the left is equivalent to adding row $j$ to row $i$. Similarly, let $P$ denote the matrix so that multiplication from the right applies a fixed permutation to the columns of $A$. Since the columns of $P$ are orthonormal, then $P^{-1} = P^T$, and thus one would write $P^TAP$ 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 $S^j_i$, 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.

$$
\begin{align*}
R_1 & = \begin{bmatrix} a & b & c \\ u & 1 & 1 \\ v & 1 & 1 \\ w & 1 & 1 \end{bmatrix} \\
V_1 & = \begin{bmatrix} a & b & c \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}
\end{align*}
\begin{align*}
S^2_1 & \rightarrow \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} & P & = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \\
S^2_1 & \rightarrow \begin{bmatrix} u & 1 & 1 \\ v & 1 & 1 \\ w & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}
\end{align*}
$$

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 $S^2_1$ 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. Informally, since column operations are the largest complexity operations needed and each column can have potentially $m$ entries, the complexity of vineyards is $O(m)$ per transposition. However, there are a number of subtleties involved in this statement. The first is that both the $V$ and $R$ matrices tend to be quite sparse, and thus column and row operations rarely require $m$ operations—in fact, as a rule of thumb, most transpositions require no column operations \cite{20}. A second subtlety, due to the necessity of sparse matrix data structures, is that querying the non-zero status of any particular entry typically takes $O(\log m)$ time. This hidden cost may seem minor, however, the quadratic scaling of the number of transpositions vineyards typically requires makes this cost non-trivial. In fact, achieving $O(m)$ complexity per transposition requires a special sparse matrix representation that allows swapping any two rows and columns in $O(1)$—see the appendix for more details and discussion. Finally, note the constant factor associated with the stated complexity: inspection of the individual cases of the algorithm from \cite{2} shows that there are at most two $O(m)$ operations, on both $R$ and $V$, that may be needed for any single transposition.

### 2.4 Moves

Busaryev et al. \cite{12} introduced an algorithm which maintains an $R = DV$ 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 $\sigma_i$ at position $i$ to position $j$. If $j = i \pm 1$, this operation is an adjacent transposition, and thus in some sense Busaryev’s move algorithm is a generalization of vineyards.

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

- **M1**: Querying the non-zero status of entries in either $R$ or $V$ occurs once per move.
- **M2**: $R = DV$ is not guaranteed to be valid during the movement of $\sigma_i$ to $\sigma_j$.

First, consider property M1. Prior to applying any permutation $P$ to the decomposition, it is necessary to remove non-zero entries in $V$ which render $P^TVP$ non-upper triangular, to maintain invariant I2. If one uses the (augmented) sparse matrix representation described in \cite{2}, checking whether these entries are non-zero takes $O(\log m)$ per column. Since moving a simplex from $i$ to $j$ using vineyards performs $|i - j| - 1$ transpositions, this query operation is required
at most \(O(|i-j|)\) times. By comparison, if a row-oriented sparse matrix implementation is used, then a move operation can perform these non-zero entry checks in just one \(O(m)\) pass, prior to performing any column operations.

Property M2 indicates move operations only guarantee the decomposition is valid after the operation completes, implying that the decomposition is not fully maintained during the execution of RestoreRight and RestoreLeft below. This is a significant difference compared to the vineyards algorithm, which was designed specifically to maintain a valid decomposition after each transposition. 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, a move operation \(Move(i, j)\) generates only one. Indeed, if one has a pair of filtrations \((K_0, K_1)\) each with \(m\) simplices, and we assume that each simplex \(\sigma_i\) switches its relative ordering with another simplex \(\sigma_j\) at most once in some ordered schedule of transpositions, then the number of intermediate persistence diagrams is bounded above by \(O(m^2)\) using transpositions, whereas with move operations this bound clearly is at most \(O(m)\).

**Example:** We continue the example used in sections 2.3 and 2.2 to illustrate moves. Consider moving the edge \(a\) to the position of edge \(c\) in the filtration. The donor columns introduced in [12] are shown on the left side of each matrix. Note that using vineyards, the equivalent permutation to the decomposition would require 4 column operations on both \(R_1\) and \(V_1\), respectively, whereas a single move operation accomplishes this permutation using only 2 column operations per matrix.

\[
\begin{align*}
\begin{bmatrix}
d_R & a & R_1 & a & b & c \\
u & 1 & u & 1 & 1 \\
v & 1 & v & 1 & 1 \\
w & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
b & a & b & c \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
c & a & b & c \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
c & b & a & c \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
R_1 & b & c & a \\
1 & 1 & \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix}
d_V & a & V_1 & a & b & c \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
b & a & b & c \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
c & a & b & c \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
c & b & a & c \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & \end{bmatrix} & \rightarrow \\
\begin{bmatrix}
V_1 & b & c & a \\
1 & 1 & \end{bmatrix}
\end{align*}
\]

**Algorithm 2** Move Right Algorithm

1. function RESTORERIGHT\((R, V, I = \{I_1, I_2, \ldots, I_s\})\)
2. \((d_{low}, d_R, d_V) \leftarrow (\text{low}_R(I_1), \text{col}_R(I_1), \text{col}_V(I_1))\)
3. for \(k \in I_2, \ldots, I_s\) do
4. \((d_{low}', d_R', d_V') \leftarrow (\text{low}_R(k), \text{col}_R(k), \text{col}_V(k))\)
5. \((\text{col}_R(k), \text{col}_V(k)) \leftarrow (d_R, d_V)\)
6. if \(d_{low}' < d_{low}\) then
7. \((d_{low}, d_R, d_V) \leftarrow (d_{low}', d_R', d_V')\)
8. return \((R, V, d_R, d_V)\)

1. function MOVERIGHT\((R, V, i, j)\)
2. \(I = \text{columns satisfying } V[i, i : j] \neq 0\)
3. \(J = \text{columns satisfying low}_R \in [i : j] \text{ and row}_R(i) \neq 0\)
4. \((R, V, d_R, d_V) \leftarrow RESTORERIGHT(R, V, I)\)
5. \((R, V) \leftarrow RESTORERIGHT(R, V, J)\)
6. \((R, V) \leftarrow (PR^T, PV^T)\)
7. \((\text{col}_R(j), \text{col}_V(j)) \leftarrow (Pd_R, Pd_V)\)
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
Let us begin with a brief overview of the pipeline, which we also outline in Algorithm 4 below. As before, we assume our contribution: Move Schedules how to apply a donor column kind of concept. Fortunately, we can still separate these two stages.

Algorithm 3 Move Left Algorithm

1: function RESTORELEFT(R, V, \( K = \{ k_1, k_2, \ldots, k_s \} \))
2: \((l, r) \leftarrow\) indices \( l, r \in K \) satisfying \( l < r \), \( \text{low}_R(l) = \text{low}_R(r) \) maximal
3: while \( \text{low}_R(l) \neq 0 \) and \( \text{low}_R(r) \neq 0 \) do
4: \((\text{col}_R(r), \text{col}_V(r)) \leftarrow (\text{col}_R(l), \text{col}_V(l))\)
5: \(K \leftarrow K \setminus l\)
6: \((l, r) \leftarrow\) indices \( l, r \in K \) satisfying \( l < r \), \( \text{low}_R(l) = \text{low}_R(r) \) maximal
7: return \((R, V)\)

Algorithm 4 is purely illustrative at this point, and is meant to serve as a guidepost before discussing its components in depth. Specifically, lines (4-5) are discussed next in section 3.2, and the algorithm for line (6) is given in section 3.3.

Although conceptually similar, there is an asymmetry between MoveRight and MoveLeft. Moving a simplex upwards in the filtration requires removing non-zero entries along several columns of a particular row in \( V \) so that the corresponding permutation does not render \( V \) non-upper triangular. The key insight of the algorithm presented in [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, \( R \) stays reduced during this cancellation procedure in all but one column, however the subsequent permutation to \( R \) requires reducing a pair of columns which may cascade into a larger chain of column operations to keep \( R \) reduced. Since these operations always occur in a left-to-right fashion, its not immediately clear how to apply a donor column kind of concept. Fortunately, we can still separate these two stages.

3 Our contribution: Move Schedules

3.1 Proposed approach

Let us begin with a brief overview of the pipeline, which we also outline in Algorithm 4 below. As before, we assume as input a discrete 1-parameter family of filtrations \( K = \{ K_1, K_2, \ldots, K_n \} \) of an abstract simplicial complex \( K \) with \( |K| = m \), and the goal being to compute their persistence diagrams. The pipeline can be summarized as follows.

Fix reindexing bijections \( f_i : K_i \rightarrow K_{i+1} \). Each \( f_i \) induces a bijection \( f_i^* : [m] \rightarrow [m] \), or equivalently, a permutation of the index set \([m] \). For each pair of filtrations \( (K_i, K_{i+1}) \), we assign labels to the simplices of \( K_i \) using the index set \([m] \) and relabel \( K_{i+1} \) accordingly using \( f_i \). Denote these permutations by \( p \) and \( q \). We compute the longest increasing subsequence (LIS) of \( q \) and use this subsequence to recover a longest common subsequence (LCS) \( LCS(p, q) \). We pass \( p, q \), and this LCS to our greedy scheduling algorithm, which returns as output an ordered set of move permutations \( S \) of minimum size, which we call a schedule. Not all valid schedules are created equal: some incur a much larger number of column operations, even if they have a minimal number of moves. Minimizing schedule costs will be addressed in Section 3.3. We proceed to computing an \( R = \text{DV} \) decomposition for the first filtration \( K_1 \), and then execute the moves stored in \( S \) in sequence, extracting their corresponding persistence diagrams along the way. During execution, we obtain a valid decomposition \( R = \text{DV} \) for each filtration \( K_1, K_2, \ldots, K_n \).

Algorithm 4 is purely illustrative at this point, and is meant to serve as a guidepost before discussing its components in depth. Specifically, lines (4-5) are discussed next in section 3.2, and the algorithm for line (6) is given in section 3.3.
no need to have access to—or explicitly store—the entire family of filtrations and the schedules between them. That is, Algorithm 4 can be modified to be completely online, keeping at most two filtrations and one decomposition in memory at any given time.

3.2 Minimizing schedule size

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

3.2.1 The Vineyards Setting

In the vineyards (i.e., continuous time) case, we are given a homotopy $F : K \times [0, 1] \rightarrow \mathbb{R}$ interpolating between a pair of filtrations $K_0, K_1$ of $K$, with simplices ordered according to $f_0 = F(\cdot, 0)$ and $f_1 = F(\cdot, 1)$, respectively. The choice of homotopy $F$ completely determines the number of adjacent transpositions that transform $K_0$ into $K_1$ via $F$, so to get a meaningful bound we must know something about $F$. If we assume that 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 set of adjacent transpositions $S_F$ that $F$ decomposes into is given by the pairs of simplices in $K$ with changes in their relative ordering:

$$S_F = \{(i, j) \mid f_0(\sigma_i) < f_0(\sigma_j) \text{ and } f_1(\sigma_i) > f_1(\sigma_j) \} \quad i, j \in [m]$$

$$S_F = \{s_1, s_2, \ldots, s_k\}$$

Let $p = [m]$ and let $q = \text{Im} f^*$, where $f^* : [m] \rightarrow [m]$ is the one-to-one correspondence induced by the reindexing bijection $f : K_0 \rightarrow K_1$. Then $S_F$ is given by the inversions between $p$ and $q$:

$$S_F = \text{Inv}(p, q) = \{(i, j) \mid p(i) < p(j) \text{ and } q(i) > q(j) \} \quad i, j \in [m] \}$$

The cardinality of $S_F$ is exactly the Kendall-$\tau$ distance $K_\tau(p, q)$ between $p$ and $q$:

$$K_\tau(p, q) = |\text{Inv}(p, q)|$$

Geometrically, our assumptions on $F$ imply that 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 $F(\sigma, \tau) = (1 - \tau)f_0(\sigma) + \tau f_1(\sigma)$, 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 [25], where $k$ is the output-sensitive term. Since $k \sim O(m^2)$ in the worst case, achieved when $f_1 = -f_0$, we have the following upper bound on $|S_F|

$$|S_F| \approx O(m^2)$$
This bound is exhibited in the grayscale image data example in section 1.2: each 9x9 patch contains (81, 208, 128) simplices of dimensions (0, 1, 2) yielding m = 417 simplices in total which, as the homotopy is simulated, generates ≈ 70,000 transpositions. The upper bound size on |S_F| is thus \( \binom{417}{2} \approx 87,000 \), which nearly occurs in this case, since the last filtration is approximately the reverse of the filtration at the beginning. This quadratic scaling can induce a number of issues in practical implementations of vineyards. Indeed, although many transpositions require no column reductions in practice, detecting the existence of non-zero entries at specific positions in the sparse matrix induces a number of issues in practical implementations of vineyards. Indeed, although many transpositions require no column reductions in practice, detecting the existence of non-zero entries at specific positions in the sparse matrix takes \( O(\log(m)) \) time. Since this check is required after each permutation is applied, it is in some sense unavoidable in the vineyards algorithm.

The ordered set of inversions \((i, j) \in S_F\) can be interpreted as permutations to apply to the decomposition, and hence can be thought of as a schedule with respect to \( F \). If our goal is to decrease the size of \(|S_F|\), one option is to coarsen \( S_F \) to a new schedule \( \tilde{S}_F \) by collapsing contiguous sequences of adjacent transpositions to moves, via the map:

\[
(i, i + 1)(i + 1, i + 2)\cdots(j - 1, j) \mapsto (j, i + 1, \cdots, j - 1, i) \quad \text{if } i < j
\]

We expect \( \tilde{F} \) to be cheaper to execute than \( S_F \), with the tradeoff that less diagrams are produced. We make this precise next:

**Proposition 1** Let \( R = DV \) be a valid decomposition of size \((m \times m)\), let \( C \) denote the number of \( O(m) \) operations required to execute schedule \( S_F \), and similarly let \( L \) be the number of \( O(m) \) operations required to execute a coarsened schedule \( \tilde{S}_F \) of \( S_F \). We have the following relationship between \( C \) and \( L \):

\[
\frac{C}{2} \leq L \leq C
\]

**Proof 1** Consider executing the vineyards algorithm on a given schedule \( S_F \). The dominant subcomputation for the vineyards algorithm are the column operations—which take \( O(n) \) time—since the permutations can be made to the matrix in \( O(1) \) time and all other computations take at most \( O(\log(m)) \) time. In the vineyards algorithm, there are at most 2 column operations for a given adjacent transposition per matrix. Thus, for any contiguous sequence of transpositions \((i, i + 1), (i + 1, i + 2), \ldots, (j - 1, j)\) we require at most \( 2(|i - j|) \) column operations in both \( R \) and \( V \), giving a total of \( 4(|i - j|) \) column operations.

Now consider using the MoveRight\((i, j)\) for a single move operation, as a replacement for the sequence of transpositions above. Here again, the dominant cost again are the column operations (line 5), since although permutations take \( O(|i - j|) \) time we require only 1 such permutation per movement. If we allocate an extra \( O(m) \) memory space during the movement, notice that lines 4, and 7 of Restore* (but not line 2) and line 7 of Move* in Algorithm 2 can be completed \( O(1) \) via a pointer swapping argument. Since a move operation requires at most 1 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. The claimed inequality follows.

In terms of bounds, we have \(|\tilde{S}_F| \leq |S_F|\), and the associated coarsened schedule \( \tilde{S}_F \) can be computed in \( O(m) \) time. However, the coarsening depends entirely on the choice of \( F \) and the upper bound from 6 remains, as its always possible that there are no contiguous subsequences to collapse. If a straight line homotopy is used, the number of collapsible subsequences may depend on the simplices categorization as critical points; transpositions between pairs of simplices which trigger changes in homology tend to be more expensive. Cohen-Steiner [2] referred to these as switches, see [20] for more details.

### 3.2.2 The Moves Setting

As we mentioned before, vineyards maintains \( j - i \) valid decompositions while performing the sequence \((i, i + 1), (i + 1, i + 2), \ldots, (j - 1, j)\) of adjacent transpositions. The benefit of move operations is that one need not maintain a valid decomposition during the movement \((i, j)\), only at the end; thus, it is worth investigating whether the vineyard bounds can be improved. In the case of vineyards, the bound is clear: if \( \kappa = K_\tau(K_0, K_1) \) is the Kendall-\( \tau \) distance, then \( \Omega(\kappa) \) adjacent transpositions are required to transform \( K_0 \to K_1 \). In contrast, if we allow move operations in any order, we have \( O(m) \) as a trivial upper bound: simply move each simplex in \( K_0 \) into its position in the filtration given by \( K_1 \) in the order given by the latter. However, it’s not immediately clear whether this bound is tight. To establish this formally, we require a few definitions.

Let \( S_m \) denote the symmetric group of permutations of \([m]\) under function composition \( \circ \). Given two fixed permutations \( p, q \in S_m \) and a set \( \Sigma \subseteq S_m \), the following are common problems in the literature:

1. Find an ordered sequence of permutations \( s_1, s_2, \ldots, s_d \in \Sigma \) whose composition transforms \( p \) into \( q \):

\[
s_d \circ \cdots \circ s_2 \circ s_1 \circ p = q
\]
2 Find a sequence satisfying (1) of minimal length \( (d) \)

3 Find the minimal length \( d \), referred to as the distance between \( p \) and \( q \) with respect to \( \Sigma \).

A sequence \( S = \{s_1, s_2, \ldots, s_d\} \subseteq \Sigma \subseteq S_m \) of operations mapping \( p \rightarrow q \) is sometimes called a sorting of \( p \). In the context of performing updates to the \( R = DV \) decomposition using moves, such sequences will form schedules. There are typically many ways to solve (1), and any solution to (2) necessarily solves (1) and (3). When \( p, q \) are interpreted as strings, distances defined with respect to a fixed \( p, q \) are typically many ways to solve (1), and any solution to (2) necessarily solves (1) and (3). When \( p, q \) are interpreted as strings, distances defined with respect to a fixed \( \Sigma \subseteq S_m \) are commonly referred to as edit distances \( [26] \), which are denoted as \( d_{\Sigma}(\cdot, \cdot) \), and the subsequent elementary operations on the strings are called edit operations. The choice of \( \Sigma \) defines the universe of operations and thus the type of edit distance being measured—otherwise if \( \Sigma = S_m \), then \( d_{\Sigma}(p, q) = 1 \) for any \( p \neq q \in S_m \).

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

\[
\begin{pmatrix}
1 \cdots i - 1 & i & i + 1 & \cdots & j - 1 & j & j + 1 & \cdots & k & k + 1 & \cdots & l - 1 & l & l + 1 & \cdots & m \\
1 \cdots i - 1 & k & k + 1 & \cdots & l - 1 & j & j + 1 & \cdots & k & i + 1 & \cdots & j - 1 & l & l + 1 & \cdots & m
\end{pmatrix}
\]

Common edit distances found in the literature expressed as specializations of \( \beta \) are given below:

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

Both the difficulty of obtaining a minimal sorting and the size of \( d_{\Sigma}(p, q) \) vary dramatically with the choice of \( \Sigma \). For example, while sorting by transpositions and reversals is NP-hard, and sorting by prefix transpositions (i.e., when \( i = 1 \) always) is unknown, there are polynomial time algorithms for sorting by block interchanges, exchanges, and prefix exchanges \( [26] \). Sorting by adjacent transpositions can be achieved in many ways: any sorting algorithm that exchanges two adjacent elements during its execution (e.g., bubble sort, insertion sort, merge sort) yields a sorting of size \( K_r(p, q) \).

Here we consider sorting by moves. In terms of permutations, a permutation \( m_{ij} \) that moves \( i \) to \( j \) in \([m]\), for \( i < j \), corresponds to:

\[
\begin{pmatrix}
1 & \cdots & i - 1 & \cdots & j - 1 & j + 1 & \cdots & m \\
1 & \cdots & i - 1 & \cdots & j - 1 & j & \cdots & m
\end{pmatrix}
\]

In the context of edit operations, observe that a move operation can be interpreted as the composition of a deletion operation followed by an insertion operation:

\[
m_{ij} = (\text{ins}_j \circ \text{del}_i)
\]

where \( \text{del}_i \) denotes the edit operation that deletes the character at position \( i \) and \( \text{ins}_j \) similarly 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 insertions and deletions, where each insertion/deletion operation is coupled. This is exactly the LCS (Longest Common Subsequence) distance. It is well known \( [27] \) that the LCS distance between two strings \( p, q \) of sizes \( m \) and \( n \), respectively, is:

\[
d_{\text{Lcs}}(p, q) = m + n - 2|\text{LCS}(p, q)|
\]

Intuitively, in the context of strings, given \( \mathcal{L} = \text{LCS}(p, q) \) one may transform \( p \rightarrow q \) by simply deleting the symbols from \( p \setminus \mathcal{L} \) and then inserting symbols from \( q \setminus \mathcal{L} \). In the context of permutations, these operations are coupled, thus to ensure only \( 2d \) such edit operations are used to transform \( p \rightarrow q \), each move operations must increase the size of \( \mathcal{L} \) monotonically. Thus, by reducing the problem to the permutation edit distance, we obtain the following lower bound on the minimum size of a sorting (i.e., a schedule) using moves.

**Proposition 2 (Schedule size)** Let \( K_0 \) and \( K_1 \) be filtrations of a simplicial complex \( K \) with \( m \) simplices. Then, a smallest size move schedule \( S^* \) transforming \( K_0 \) into \( K_1 \) has cardinality

\[
d = m - |\text{LCS}(K_0, K_1)|
\]

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

\footnote{In comparative genomics, exchanging two elements is sometimes called a super short transposition.}
Proof 2 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.
\[
\text{Conceptually, left-invariance implies that the edit distance between any pair of permutations } p, q \text{ is invariant under an arbitrary relabeling } p, q - \text{as long as the relabeling is consistent. Thus, the following identity always holds:}
\]
\[
d_\Sigma(p, q) = d_\Sigma(p \circ r, q \circ r) \quad \text{for all } p, q, r \in S_m
\]
where \( \iota = [m] \), the identity permutation. Suppose we are given permutations \( p = K_0, q = K_1 \), and we seek to compute \( \text{LCS}(p, q) \). Consider the permutation \( p' = q^{-1} \circ p \). Since the LCS distance is a valid edit distance, if \( |\text{LCS}(p, q)| = k \), then \( |\text{LCS}(p', \iota)| = k \) as well. Notice that \( \iota \) is strictly increasing and that any common subsequence \( \iota \) has with \( p' \) must also be strictly increasing. Thus, the problem of computing \( \text{LCS}(p, q) \) 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 \([28]\).

The optimality of \( d \) follows from the optimality of the well-studied LCS problem \([29]\).

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

Proof 3 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:
\[
\mathbb{E} \ L(p) = \ell_m = \frac{1}{m!} \sum_{p \in S_m} L(p)
\]

The history of the mathematics of estimating this quantity dates back at least 50 years, and is sometimes called the Ulam-Hammersley problem. The seminal work by Baik et al. \([30]\) established that as \( m \to \infty \):
\[
\ell_m = 2\sqrt{m} + cm^{1/6} + o(m^{1/6})
\]
where \( c = -1.77108 \ldots \). As a result, we have:
\[
\frac{\ell_m}{\sqrt{m}} \to 2 \quad \text{as } m \to \infty
\]
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 \( \text{LCS}(p, q) \Leftrightarrow \text{LIS}(p') \), the claimed bound follows.

The upper bound from Corollary 1 captures the size of \( S^* \) under worst-case conditions. It is worth noting that this bound applies to pairs of uniformly sampled permutations, as opposed to uniformly sampled filtrations—simplicial filtrations have more structure due to the subcomplex relation that must be respected. Thus, while \( O(m - \sqrt{m}) \) in expectation is an improvement over \( O(m) \), it is still not tight. However, Boissonnat \([31]\) has shown that for any \( d \)-dimensional simplicial complex \( K \) with \( m \) simplices, the number of distinct filtrations built from \( K \) is at least \( \lfloor \frac{e+1}{d+1} \rfloor \), where \( t \) is the number of distinct filtration values associated with the complex \( K \). In the case where \( t \sim O(m) \) and \( d << m \) fixed, this lower bound grows similarly to \( m! \)—thus, \( O(m - \sqrt{m}) \) is not too pessimistic of an upper bound when the pair of filtrations are no more similar than two random permutations. In practice, when one has a time-varying filtration and the sampling points are relatively close [in time], the LCS between adjacent filtrations is expected to be much larger, subsequently making \( d \) much smaller. For example, the average size of the LCS across the 10 filtrations from Section 1.2 each having \( m = 417 \) simplices sampled at evenly spaced time points was 343, implying \( d \approx 70 \) permutations needed on average to update the decomposition at adjacent time points.

Constructing a sorting

In the previous section, we showed how computing the LCS between two permutations \( p, q \in S_m \) reduces to computing the LIS of a single permutation \( p' \), and we outlined how to do this in \( O(m \log \log m) \) time. It is not clear, however, that given \( L = \text{LIS}(p') \), one can obtain a sorting \( p \mapsto q \) in an efficient way. We outline below a simple procedure which constructs such a sorting \( S = (s_1, \ldots, s_d) \) in \( O(dm \log m) \) time.

Before introducing the algorithm, we require a few definitions. Recall that a sorting \( S \) with respect to two permutations \( p, q \in S_m \) is an ordered sequence of permutations \( (s_1, s_2, \ldots, s_d) \) whose product under composition with \( p \) yields \( q \), i.e.
\[
q = s_d \circ \ldots \circ s_1 \circ p
\]
We call such a sorting a valid sorting. To do this efficiently, we need a means of choosing a permutation \( s \) that satisfies equation 9. Let \( T \) denote an ordered set-like data structure that supports the following operations on elements \( p \) and \( q \) within the set \( M \): 

\[
\text{LCS}(s \circ p, q) = |\text{LCS}(p, q)| + 1
\]

By definition, a subsequence \( L \) of symbols common to both \( p \) and \( q \) satisfies:

\[
p^{-1}(\sigma) < p^{-1}(\tau) \quad \Rightarrow \quad q^{-1}(\sigma) < q^{-1}(\tau) \quad \forall \sigma, \tau \in L
\]

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

\[
|\text{LCS}(s \circ p, q)| = |\text{LCS}(p, q)| + 1
\]

The problem of constructing a sorting \( S \) of size \( d \) thus reduces to the problem of choosing a sequence of \( d \) valid moves. We call such a sorting a valid sorting. To do this efficiently, we need a means of choosing a permutation \( s \in S_m \) that satisfies equation 9. Let \( T \) denote an ordered set-like data structure that supports the following operations on elements \( \sigma \in M \) from the set \( M = \{0, 1, \ldots, m + 1\} \):

1. \( T \cup \sigma \)--inserts \( \sigma \) into \( T \),
2. \( T \setminus \sigma \)--removes \( \sigma \) from \( T \),
3. \( T_{\text{succ}}(\sigma) \)--obtain the successor of \( \sigma \) in \( T \), if it exists, otherwise return \( m + 1 \)
4. \( T_{\text{pred}}(\sigma) \)--obtain the predecessor of \( \sigma \) in \( T \), if it exists, otherwise return \( 0 \)

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

\[
p^{-1}(T_{\text{pred}}(\sigma)) < p^{-1}(\sigma) < p^{-1}(T_{\text{succ}}(\sigma))
\]

for every \( \sigma \in T \). Now, suppose we choose a symbol \( \sigma \notin T \). If \( p^{-1}(\sigma) < p^{-1}(T_{\text{pred}}(\sigma)) \), then we must move \( \sigma \) to the right in \( p \) such that equation 10 holds. Similarly, if \( p^{-1}(T_{\text{succ}}(\sigma)) < p^{-1}(\sigma) \), then we must move \( \sigma \) left in \( p \) to increase the size of the LCS. Assuming the structure \( T \) 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 is shown in pseudocode in Algorithm 5.

### 3.3 Minimizing schedule cost

#### 3.3.1 Motivation

The results from the previous section yield a sufficient algorithm for generating move schedules of minimal cardinality: simply compute a LCS(\( K_0, K_1 \)) for a pair \( (K_0, K_1) \) and fix an order of simplices in the set LCS(\( K_0, K_1 \) \( \setminus \) \( K_0 \)) to move. Any sequence of moves which increases the LCS on each move is guaranteed to have size \( m - |\text{LCS}(K_0, K_1)| \), and the reduction to the permutation edit distance problem ensures this size is optimal. However, like in the vineyards algorithm, certain pairs of simplices cost more to exchange depending on whether they are critical pairs in the sense described in 2. This small variation in cost-per-transposition induces a large variability in the cost of randomly generated schedules, which is computationally undesirable in practice. Ideally, we would like to generate a schedule which not only is minimal in size, but is also minimal in terms of the column operations it requires.

| Algorithm 5 Sorting algorithm |
|---|
| 1: function LCS-SORT(p, q) |
| 2: \( \bar{p} \leftarrow q^{-1} \circ p \) |
| 3: \( L \leftarrow \text{LCS}(p, q) = \text{LIS}(\bar{p}) \) |
| 4: \( (S, D, T) \leftarrow (\emptyset, \bar{p} \setminus L, L) \)
| 5: while \( D \) is not empty do |
| 6: \( \sigma \leftarrow \text{an arbitrary element in } D \)
| 7: \( (i, i_p, i_n) \leftarrow (p^{-1}(\sigma), \tau_p^{-1}(\text{pred}(\sigma)), \tau_n^{-1}(\text{succ}(\sigma))) \) |
| 8: if \( i < i_p \) then |
| 9: \( j \leftarrow \text{arbitrary element in } [i, i_n] \)
| 10: else \( i_n < i \)
| 11: \( j \leftarrow \text{arbitrary element in } (i_p, i_n) \)
| 12: \( (S, D, T) \leftarrow (S \cup (i, j), D \setminus \sigma, T \cup \sigma) \) |
| 13: \( \bar{p}^{-1} \leftarrow \bar{p}^{-1} \circ m_{ij}^{-1} \) |
| 14: return \( S \) |
3.3.2 Greedy approach

One advantage of using move operations is that the cost of applying a permutation to some given \( R = DV \) decomposition can be determined efficiently prior to performing any column operations. Specifically, the cost of determining \(|I| + |J|\) in lines (2) and (3) of Algorithm cannot take more than \( O(m) \) time. Towards minimizing the number of column operations a schedule requires, 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 filtration contains \( m = 10 \) simplices comprising the 1-skeletons of a 3-simplex with a particular choice of ordering. Relabeling the simplices of \( K_0 \) to the index set \([m]\) and modifying the labels of \( K_1 \) correspondingly yields permutations given below:

\[
K_0 = \{a \ b \ c \ d \ u \ v \ w \ x \ y \ z\} = 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10
\]
\[
K_1 = \{a \ b \ c \ d \ x \ y \ z \ u \ v \ w\} = 1 \ 2 \ 3 \ 4 \ 8 \ 9 \ 10 \ 5 \ 6 \ 7
\]

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

\[
S_1 = m_{zu}, m_{yu}, m_{zu}, S_3 = m_{yu}, m_{xy}, m_{zu}, S_5 = m_{zu}, m_{xz}, m_{yz}
\]
\[
S_2 = m_{zu}, m_{zu}, m_{yz}, S_4 = m_{yu}, m_{zu}, m_{xy}, S_6 = m_{zu}, m_{yz}, m_{zz}
\]

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. A greedy strategy which always selects the cheapest move in succession would begin by moving \( x \) or \( z \) first, since these are the cheapest moves available, which implies one of \( S_1, S_2, S_5, S_6 \) would be picked depending on the choice of a tie-breaker. While the cheapest schedule \( S_1 \) is in this candidate set, so is \( S_6 \), the most expensive schedule. As a result, a greedy strategy which chooses the lowest-cost move may not yield an optimal schedule.

### 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    |

3.3.3 Proxy objective

Although the greedy approach from the last section can lead to arbitrarily high cost schedules, similar greedy-like strategies can yield computationally efficient heuristics in practice, even if they are suboptimal. For example, in the previous section choosing the lowest-cost move takes \( \approx O(dm) \)—implying a greedy strategy would require \( O(d^2m) \) time—which is computationally viable as a heuristic when \( d << m \). However, when \( d \sim O(m) \), the sorting procedure shown using Algorithm approaches \( O(m^2 \log m) \). We seek a fast procedure for generating schedules; if the schedule generation procedure exhibits a complexity of \( \approx O(m^2) \) or higher, then such a procedure would offer no computational benefit compared to the reduction algorithm. Indeed, observe the vineyards algorithm requires \( O(m \log m + k) \) preprocessing time to compute a schedule of size \( k \sim O(m^2) \) transpositions, thus for Algorithm to be considered a more efficient alternative to vineyards, a solution which is subquadratic in \( m \) is desirable.

In light of the discussion above, we consider whether there exists an alternative proxy objective to optimize that is both related to the cost of a schedule and not dependent on the entries in the decomposition. Obviously, the number of column operations needed to move a simplex from position \( i \) in the filtration to position \( j \) is upper bounded by \(|i - j|\). In practice, due to the sparsity of both \( R \) and \( V \), this bound is very loose—the number of column operations is generally much smaller. Nonetheless, the complexities of the move operations discussed in Section are dependent on the magnitude of \(|i - j|\), and thus it is worth considering whether minimizing the total displacement would be an effective proxy objective.
If one forms bijections between two given filtrations \( p : K_0 \to [m], q : K_1 \to [m] \) to permutations of \([m]\) as before, then the Kendall-\(\tau\) distance yields an upper bound on the number of column reductions needed to perform any schedule. This follows from the fact that any inversion between \(p\) and \(q\) may induce a column operation, and the number of inversions between \(p\) and \(q\) is exactly \(K_\tau(p, q)\). Another common distance for measuring the disarrangement between permutations is the Spearman distance, defined as:

\[
F(p, q) = \sum_{i\in[m]} |p^{-1}(i) - q^{-1}(i)|
\]  

(11)

Intuitively, the Spearman distance can be interpreted as \(l_1\)-type distance between two permutations. Like the Kendall-\(\tau\) distance, it is a metric, it is invariant under relabeling, and the distance between pairs of uniformly random permutations \(p, q \sim S_m\) are asymptotically normal. Indeed, Diaconis et al. [32] showed the the Spearman distance between any two permutation \(p, q \in S_m\) approximates their Kendall-\(\tau\) distance within a factor of two:

\[
K_\tau(p, q) \leq F(p, q) \leq 2K_\tau(p, q)
\]  

(12)

Although very similar to the Kendall-\(\tau\) distance, the Spearman distance conveys certain computational advantages not inherited by the Kendall distance due to its simplicity. For example, the Spearman distance can be computed in \(O(m)\) time.

Next we consider the difficulty of obtaining a sorting \(S\) of size \(d = m - \text{LCS}(p, q)\) which has minimal total displacement. That is, incorporating the Spearman distance in the proxy objective. Formally, given \(p, q \in S_m\) and some sorting \(S = (s_1, \ldots, s_d)\), let \(s_i = s_i \circ \cdots \circ s_2 \circ s_1\) denote the composition of the first \(i\) permutations in \(S\), where \(i < d\). To incorporate the notion of displacement as an additive minimization problem, we consider minimizing an upper bound on the Spearman distance by decomposing it into a sum of \(d - 1\) local distances:

\[
F(p, q) \leq \sum_{i=1}^{d-1} F(s_i \circ p, s_{i+1} \circ p)
\]  

(13)

where equality is only achieved with schedules \(S\) for which the displacement of each symbol between its initial position in \(p\) and its final position in \(q\) is non-increasing with every application of \(s_i\). Note this not guaranteed if \(S\) was derived using the strategy from section[3]

The problem of minimizing the right hand side of equation[13] 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. If the \((p, m_{ij} \circ p)\) are drawn as a bipartite graph, with the elements of \(p\) in one layer and the elements of \(m_{ij} \circ p\) in another, then observe that \(F(p, m_{ij} \circ p)\) is twice the number of edge crossings in the graph. We given an example below illustrating this.

**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 Figure[2]

![Figure 2: Three sortings, \(S_1, S_2,\) and \(S_3\), with 21, 31, and 35 crossings, resp.](image)

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 \(\text{LCS}(p, q) = (4 7 8)\) and all three schedules transform \(p \mapsto q\) 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 general, the $k$-layer crossing minimization problem for $k$ sets of permutations is NP-hard for $k \geq 4$ [33]. As a result, we do not expect there to be a polynomial time algorithm for optimally achieving this proxy objective. However, the simplicity of both the Spearman distance and the types of permutations we consider here enables certain computational advantages. Suppose we begin with an array $A$ 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 the execution of Algorithm 5 we may obtain $F(p,q)$ simply by having access to the sum of every entry in $A$. There are two degrees of freedom in Algorithm 5: the first is in the choice of $\sigma \in D$, and the second is in choosing $j$. If we fix a heuristic for the latter, then we have a set of possible valid moves $S_D$ induced by each $\sigma \in D$ to choose from. Observe that each permutation $s_\sigma \in S_D$ changes the displacement of every symbol in at most three different ways:

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

That is, if $s_\sigma$ represents a move permutation moving $\sigma$ from position $i$ to position $j$, then relative to the previous permutation $p$, the displacement of $\sigma$ in $s_\sigma \circ p$ changes by $|i - j|$ and the displacement of any symbol between $i$ and $j$ changes by $\pm 1$. All other symbol displacements are unchanged. Thus, if we replace $A$ with a data structure supporting $O(\log m)$ access time to aggregate information (i.e. the sum) and $O(\log m)$ modification time ability on $|i - j|$ entries (simultaneously), we could greedily choose the next permutation $s$ to maximize:

$$s_{\text{greedy}} = \arg \min_{s \in S_D} F(s \circ p, q)$$

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 $O(\log m)$. It is known that one can apply constant-factor updates to multiple values in a prefix sum in $O(\log m)$ time by any prefix-sum data structure that also supports range updates—e.g. an implicit treap. 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 a greedy solution to equation 14 can be computed in $O(d^2 \log m)$ time.

4 Empirical results

Crocker stacks

There are many challenges to depicting topological behavior in dynamic settings. One approach is to trace out the curves constituting a continuous family of persistence diagrams in $\mathbb{R}^3$—the vineyards approach—however this visualization can be cumbersome to work with as there are potentially many such vines tangled together, making topological critical events with low persistence difficult to detect. Moreover, the vineyards visualization does not admit a natural simplification utilizing the stability properties of persistence, as individual vines are not stable: if two vines move near each other and then pull apart without touching, then a pairing in their corresponding persistence diagrams may cross under a small perturbation, signaling the presence of an erroneous topological critical event [14, 15].

Acknowledging this, Topaz et al. [14] proposed the use of a 2-dimensional summary visualization, called a crocker plot. In brief, a crocker plot is a contour plot on a family of Betti curves. Formally, given a filtration $K = K_0 \subseteq K_1 \subseteq \cdots \subseteq K_m$, a $p$-dimensional Betti curve $\beta_p^*(\tau)$ is defined as the ordered sequence of $p$-th dimensional Betti numbers:

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

Given a time-varying filtration $K(\tau)$, a crocker plot can be interpreted as a contour plot on the 1-parameter family of Betti curves $\beta_p^*(\tau)$. 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 any of the specialized methods discussed in [13]. 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.

---

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

Acknowledging this, Xian et al. [15] 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 $\alpha$-smoothed crocker plot for $\alpha \geq 0$ is the rank of the map $M_t(\epsilon - \alpha) \rightarrow M_t(\epsilon + \alpha)$ at time $t$ and scale $\epsilon$. For example, the standard crock plot is a 0-smoothed crocker plot. Allowing all three parameters $(t, \epsilon, \alpha)$ to vary continuously leads to 3D visualization called an $\alpha$-smoothed crocker stack.

Definition 1 (crocker stack) A crocker stack is a family of $\alpha$-smoothed crocker plots which summarizes the topological information of a time-varying persistence module $M$ via the function $f_M: [0,T] \times [0,\infty) \times [0,\infty) \rightarrow \mathbb{N}$, where:

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

A crocker stack is a sequence of $\alpha$-smoothed crocker plots that vary over $\alpha \geq 0$, satisfying $f_M(t, \epsilon, \alpha) \leq f_M(t, \epsilon, \alpha')$ for all $\alpha \geq \alpha'$. Note that, unlike crocker plots, applying this $\alpha$ 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 [15]. 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 [14] 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 naively 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 (i.e. vineyards) are equivalent to each other in the sense that either one contains the information needed to reconstruct the other [15]. Thus, computing crocker stacks reduces to computing the persistence of a (time-varying) family of filtrations.

Multidimensional persistence

Given a procedure to filter a space in multiple dimensions simultaneously, a multi-filtration, the goal of multidimensional persistence is to identify persistent features by examining the entire multi-filtration. Such a generalization has appeared naturally in many application contexts, showing potential as a tool for exploratory data analysis [34]. Indeed, one of the often noted drawbacks of persistence is its instability with respect to strong outliers. Although persistence diagrams are stable with respect to their input—in the Hausdorff sense—and thus are robust to Hausdorff noise, the
Figure 4: The cumulative number of $O(m)$ column operations (left, log-scale) of the three approaches tested and the distribution of the normalized Kendall distance between adjacent filtration pairs. The histogram on the right depicts the coarseness of the discretization; most filtration pairs have $\approx 5\%$ of their $\approx O(m^2)$ simplex pairs inverted between adjacent time steps.

The presence of strong outliers can artificially modify the connectivity of the underlying filtration, obscuring the detection of what would otherwise be significant topological structures [35]. By filtering the data according to a notion of density, in tandem with distance predicates, multidimensional persistence can enable the practitioner to handle outliers directly.

Unfortunately, multidimensional persistent homology comes with its own set of computational and theoretical challenges. There is not complete discrete invariant for multidimensional persistence—like the 1D persistence diagram—and the computations associated to other incomplete invariants exhibit high algorithmic complexity. A number of computational strategies have been proposed to counter this, such as homology preserving multifiltration simplification or parallelization schemes using shared-memory computation [36]. Rather than attempting to tackle multiple dimensions simultaneously, another approach is to focus on a fixed dimension, such as 2-D persistence. Utilizing the equivalence between the rank and fibered barcode invariants, Lesnick and Wright [10] developed an elegant way of visualizing the former via a reparameterization of the latter using standard point-line duality. This clever reparameterization effectively reduces the fibered barcode computation to a sequence of 1-D barcode computations at “template points” lying within the 2-cells of particular planar subdivision of the half-plane $[0, \infty) \times \mathbb{R}$. Indeed, since algorithm 4 was designed for precisely such a computation, the algorithm proposed by [10] is prototypical of the class of methods that stand to benefit from move scheduling.

The purpose of the following sections is to uncover the intrinsic bottlenecks impeding the computation of the fibered barcode invariant in practical settings. Towards this goal, we briefly summarize the theory established by several researchers [10, 37, 38], focusing on their computational and algorithmic details. Our goal is to show that the highest complexity subcomputation of the fibered barcode invariant algorithm proposed by [10] is precisely the type of computation we focus on in this effort. In doing so, we substantiate our claims from section 1.2 that our scheduling approach can improve the efficiency of a real-world application of multidimensional persistence by an order of magnitude by comparing Algorithm 4 to several alternatives on a real-world data set and problem: the problem of identifying the underlying topological space of a data set extracted from high-contrast natural images.

**The Fibered Barcode**

Lesnick and Wright [10] associate three simple invariants to a 2-dimensional persistence module $M$: the dimension function, the multigraded Betti numbers, and the fibered barcode. The dimension function is simply the function
which maps every point \( a \in \mathbb{R}^2 \) to \( \dim(M_a) \). It is a simple and easy to visualize invariant, but is unstable and yields no information about the persistent features of \( M \). The multigraded Betti numbers constitute a natural and important class of invariants often studied in algebraic geometry and commutative algebra, though both their theory and computation is beyond the scope of this effort—the interested reader is referred to [10] [39] for more details. 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 number of elements at each degree in a basis of the \( i \)-th module in a free resolution for \( M \). The fibered barcode, denoted as \( B(M) \), is defined as follows:

**Definition 2 (Fibered barcode)** The fibered barcode \( B(M) \) 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 \( B_L(M) \):

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

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

Although the fiber barcode is a very intuitive invariant, it is not clear how one might go about computing it efficiently. One could compute the 1-D persistence associated to some fixed linear combination of two given filter functions; each linear combination parameterizes a line \( L \in \mathbb{R} \times \mathbb{R}^+ \) which \( M \) restricts too. However, this is an \( O(m^3) \) time computation, which is prohibitive for exploratory data analysis purposes. Ideally, one would want to continuously vary \( L \in \mathbb{R} \times \mathbb{R}^+ \) interactively, inspecting the corresponding changes to the barcodes immediately. To enable this, [10] introduced an elegant reparameterization that not only fully characterizes \( B(M) \), but also yields an efficient method for rendering \( B_L(M) \) in real time. In the next section, we briefly summarize this reparameterization.

**Reparameterization using point-line duality**

Let \( \mathcal{L} \) denote the collection of all lines in \( \mathbb{R}^2 \) with non-negative slope, \( \mathcal{L} \subseteq \mathcal{Z} \) the collection of all lines in \( \mathbb{R}^2 \) with non-negative, finite slope, and \( \mathcal{L}' \) the collection of all affine lines in \( \mathbb{R}^2 \) with positive, finite slope. There is a standard point-line duality that gives a parameterization of \( \mathcal{L} \) with the half-plane \([0, \infty) \times \mathbb{R} \) that is convenient to work with. Define the line and point dual transforms \( D_L \) and \( D_p \), respectively, as follows:

\[
D_L : \mathcal{L} \to [0, \infty) \times \mathbb{R} \quad D_p : [0, \infty) \times \mathbb{R} \to \mathcal{L}
\]

\[
y = ax + b \mapsto (a, -b) \quad (c, d) \mapsto y = cx - d
\]

The transforms \( D_L \) and \( D_p \) 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 \( D_L(L) \in D_p(a) \). Now, for some fixed line \( L \), define the push map \( \text{push}_L(a) : \mathbb{R}^2 \to L \cup \infty \) as:

\[
\text{push}_L(a) = \min\{v \in L \mid a \leq v\}
\]

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

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

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

**Definition 3 (Critical Lines)** For some fixed \( S \subset \mathbb{R}^2 \), a line \( L \in \mathcal{L} \) is defined to be regular if there is an open ball \( B \in \mathcal{L} \) 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 \( \text{push}_L \) 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.

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 \( B_L(M) \), for any \( L \in \mathcal{L} \), in terms of a set of barcode templates \( T \) computed at every 2-cell in \( \mathcal{A}(M) \). More formally, for any line \( L \in \mathcal{L} \) and
Given bipersistence module \( M \), the three dominant computations needed to compute the fibered barcode using the reparameterization discussed above are the following:

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

Computing (1) takes approximately \( \approx O(m^3) \) using a matrix algorithm similar to Algorithm 1 [40]. Constructing and storing the line arrangement \( \mathcal{A}(M) \) with \( n \) lines and \( k \) vertices is related to the line segment intersection problem, which known algorithms in computational geometry can solve in (optimal) output-sensitive \( O((n + k) \log n) \) time (see [41] for an overview). In terms of memory complexity, the number of 2-cells in \( \mathcal{A}(M) \) is upper bounded by \( O(\kappa^2) \), where \( \kappa \) is a coarseness parameter associated with the computation of \( \beta(M) \). By combining this bound with a bound on the number of transpositions needed to traverse a particular sequence of 2-cells in \( \mathcal{A}(M) \), the analysis from [10] shows that the computation of the barcodes templates \( T \) from step (3) using vineyard updates requires on the order of \( O(m^3 \kappa + mn^2 \log \kappa) \) elementary operations and \( O(mn^2) \) storage.

Of the three steps above, the barcode template computation is both the highest complexity and most demanding computation in practice. The number of 2-cells in \( \mathcal{A}(M) \) is on the order \( O(\kappa^2) \), and \( \kappa \) itself is on the order of \( O(m^2) \) in the worst case. This suggests the computation of the barcode templates is on the order \( O(m^3) \), which is effectively intractable for even moderately-sized filtrations. In practice, the external stability result from [38] justifies the use of a simple procedure which approximates the module \( M \) with a smaller module \( M' \) using a grid-like coarsening procedure. This coarsening procedure allows the practitioner to restrict the size of \( \kappa \) to a relatively small constant, which in-turn dramatically reduces the size of \( \mathcal{A}(M) \).

There are several approaches one can use to compute \( T \), the simplest being to run Algorithm 1 independently on the 1-D filtration induced by the duals of some set of points (e.g. the barycenters) lying in the interior of the 2-cells of \( \mathcal{A}(M) \). The approach taken by [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 \( T^c \), and the computation of \( T \) now reduces to computing a path \( \Gamma \) on \( G \) which visits each vertex at least once. To minimize the computation time, assume the \( n \) edges of \( G \) are endowed with non-negative weights \( W = w_1, w_2, \ldots, w_n \) whose values \( w_i \in \mathbb{R}_+ \) represent some notion of distance. The optimal path \( \Gamma^* \) is the one of minimal length with respect to \( W \) which visits every vertex of \( G \) at least once. There is a known \( \frac{2}{5} \)-approximation that can be computed efficiently which reduces the problem to the traveling salesmen problem on a metric graph [42].

**Noisy circle example** We include a small example to illustrate many of the concepts and definitions discussed so far. Consider a small set of points distributed noisily around \( S^1 \) that contains a few strong outliers, shown on the left side of Figure 5. Filtering this data set with respect to the Rips parameter and the complement of a kernel density estimate yields a bifiltration whose dimension function and bigraded Betti numbers are shown in the middle figure. The gray areas indicate the region where the dimension function is constant; the lighter gray area where \( \dim_1(M) = 1 \), indicating where a persistent loop was detected. On the right side, the space dual to the space in the middle is shown. The black lines represent the lines of \( \mathcal{A}(M) \), the blue dashed-lines the edges of the dual graph \( G \), and the orange points the barycenters of the 2-cells in \( \mathcal{A}(M) \), depicting where the barcodes templates \( T^c \) are parameterized. The shortest path \( \Gamma^* \) spanning the vertices of \( G \) is shown by the solid rainbow colored lines, with the traversal beginning in the top-left cell (at the red edge) and ending at the bottom-right cell (the indigo edge). Traversing \( \Gamma^* \) can be done via a simple depth-first search, though note that a \( R = DV \) decomposition needs to be stored at each degree-3 or higher vertex in \( \Gamma^* \) to avoid redundant computations as the traversal recurses.
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 Hilbert function (gray), and a line \( L \) emphasizing the persistence of features with high density. On the right, the line arrangement \( A(M) \) lying in the dual space \( D_p(\alpha) \) derived from the \( \beta(M) \).

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. In the middle, the bigraded Betti numbers, the Hilbert function, and a chosen line \( L \) are shown for the coarsened grid. As before, the 0/1/2 dimension bigraded Betti numbers are shown in green/red/yellow, respectively. The blue regions highlights where \( \dim(M) = 5 \). On the right, the persistence diagram of \( M \) restricted to \( L \), with larger points indicating a multiplicity of 2. Observe five persistent features are revealed in both the middle and right plots, matching \( \beta_1 \) of the three-circle model.

### Natural images dataset

In the previous section, we outlined the computational theory of multidimensional persistence, including its relevance to our proposed move scheduling approach. In this section, we demonstrate the efficiency of our method on practical use case of multidimensional persistence: detecting the presence of a low-dimensional topological space which well-approximates the distribution of texture patches extracted from real-world image data.

A common hypothesis is that high dimensional data tend to lie in the vicinity of an embedded, low dimensional manifold or topological space. An exemplary demonstration of this is given in [43], who explored the space of 3 × 3 high-contrast patches extracted from Hans van Hateren’s still image collection [44], which consists of approximately 4,000 monochrome images depicting various areas outside around Groningen (Holland). Motivated by applications to sparse coding applications, Lee et al. [43] was interested in how these natural image patches were distributed, in pixel-space, with respect to predicted spaces and manifolds. For example, one of the motivating questions of their

---

9See [http://bethgelab.org/datasets/vanhateren/](http://bethgelab.org/datasets/vanhateren/) for details on the image collection.
research was whether there were any clear qualitative differences the distributions of patches extracted from from images of different modalities, e.g. optical versus range images. A major result from their work is that the majority of high-contrast image patches are concentrated around a 2-dimensional submanifold embedded in \( S^7 \). Supplemental experiments conducted by Carlsson et al. [45] using persistent homology confirmed that the distribution of high-contrast \( 3 \times 3 \) patches is indeed well-approximated by a Klein bottle—around 60% of the high-contrast patches from the still image data set are concentrated within a small neighborhood around a model of a Klein bottle which accounts for \( \approx 21\% \) of the volume \( S^7 \), compared to the \( \approx 84\% \) of volume of \( S^7 \) accounted for by the entire data set. Along a similar vein, in the context of sparse coding, Perea et al. [46] introduced a dictionary learning framework for estimating and representing the distribution of patches from texture images.

If one was not aware of the analysis done by [43] [44] [45] [49], it is not immediately clear a priori that the Klein bottle model is indeed a good candidate for capturing non-linearity of image patches. Nonetheless, as we have mentioned, among the first uses of persistent homology was the so-called “homology inference problem” [47]; given a suitable treatment of strong outliers using by e.g. filtering both on the geometry and the density of the image patches, the homology of the Klein bottle should become apparent through a persistence computation.

Consider the (coarsened) fibered barcode computed from a standard Rips / codensity bifiltration on a representative sample of the image data from [44] (details to follow), shown in Figure 6. Upon inspection of the bigraded Betti number and the dimension function, one finds that a large area of dimension function is constant (highlighted as the blue portion in the middle of Figure 6), wherein the first Betti number is 5. There are many spaces whose meets this criterion, though upon further inspection, one candidate space that seems particularly plausible is the three-circle model \( C_3 \). The \( C_3 \) model consists of three circles, two of which (say, \( S_v \) and \( S_h \)) intersect the third (say, \( S_{lin} \)) in exactly two points, but themselves do not intersect. Upon performing an analysis on the density of the patches analogous to the one done by [43], one finds that the mean-centered and contrast normalized \( 3 \times 3 \) patches like on a 7-dimensional ellipsoid \( S^7 \subset \mathbb{R}^9 \). Contrast is typically measured using a discrete version of the Dirichlet semi-norm, or the “\( D \)-norm” \( \|x\|_D \) for short, which is a unique scale-invariant norm on images:

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

where \( D \) is a fixed matrix which upon application to an image \( x \in \mathbb{R}^9 \) yields a value proportional to the sum of the differences between the 4-connected neighbors (described by the relation \( i \sim j \)). It is advantageous to “whiten” the data via a change of coordinates to Euclidean sphere. This can be achieved by expressing the patch data as an expansion of a certain convenient basis of 8 non-constant eigenvectors, often called the Discrete Cosine Transform (DCT) basis [43], which are given below:

![Discrete Cosine Transform (DCT) basis]

Projecting the image data onto the first two basis vectors of leads to the projection shown in the top left of Figure 6. 15 landmark points are also shown. Observe the data are distributed well around three “circles”—the outside circle capturing the rotation gradient of the image patches (\( S_{lin} \)), and the other two circles capturing the vertical and horizontal gradients (\( S_v \) and \( S_h \), respectively). Since the three circle model is the 1-skeleton of the Klein bottle [45], one may conclude that the Klein bottle might be a reasonable candidate upon which the image data are distributed. This can be further verified empirically by e.g. using the iterative, meanshift-like procedure described in [45] to fit a parameterization of the model in question to the data.

Using the same high-contrast patch data set studied in [43], we evaluate the performance of various methods at computing the fibered barcode invariant using the parameterization from [4]. Due to the aforementioned high complexity of the fibered barcode computation, we begin by working with a subset of the image patch data \( X \). Specifically, we combine the use furthest-point sampling and proportionate allocation (stratified) sampling to sample landmarks \( X \subset X \) distributed within \( n = 25 \) strata. Each strata consists of the \((1/n)\)-thick level set given by \( \rho_{15\%} \), the density estimator. The use of furthest-point sampling gives us certain coverage guarantees that the geometry is approximately preserved within each level set, whereas the stratification ensures the original density of is approximated preserved as well. From this data set, we construct a Rips-(co)density bifiltration using \( \rho_{15\%} \) equipped with the geodesic metric computed over the same \( k = 15 \) km 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 pfco1, vineyards, and our moves approach. The

![Discrete Cosine Transform (DCT) basis]
results are summarized in Table 2. We also record the number of 2-cells encountered along the path \( \Gamma \) permutations that were applied throughout the computation of the barcode templates for both vineyards and moves, denoted in the table as \( d_K \) and \( d_{LCS} \), respectively.

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

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

5 Conclusion and Future Work

In conclusion, we presented a scheduling algorithm for efficiently updating a decomposition in 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 example application of generating crocker stacks and simulating multidimensional persistence. As discussed in sections 1.3 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 envision our approach as potentially useful to any situation where the structure of interest is a parameterized family of filtrations and the goal involves computing the persistent homology for each filtration in the family. Areas of particular interest include time-series analysis and dynamic metric spaces [3].

The simple and combinatorial nature of our approach does pose some limitations to its applicability. For example, there may be better bounds or algorithms available if stronger assumptions can be made on how the filtration is changing with time. Moreover, if the filtration \( K_0 \) shares little similarity to the “target” filtration \( K_1 \), then the overhead of reducing the simplices from \( K_1 \setminus K_0 \) appended to the decomposition derived from \( K_0 \) may be large enough to motivate simply computing the decomposition at \( K_1 \) independently, especially if parallel processors are available. That is, our approach is primarily useful if each adjacent filtration in the ordered family of filtration is “close.” Another limitation to the strategy we have here is that we rely completely on the reduction algorithm for appending new simplices to the filtration. If the dynamic filtration is constantly both permanently removing and appending new simplices, then the size of the LIS may very well be on the order \( O(m - \sqrt{m}) \), rendering the efficiencies gained using our approach as negligible. Note that this overhead does not exist for the multidimensional persistence use-case: the entire decomposition is needed at each template 2-cell.

From an implementation perspective, one non-trivial complication of our approach is its heavy dependence on a particular sparse matrix data structure which permits permuting both the row and columns of a given matrix in at most \( O(m) \) time. Our use of this data structure is motivated by the fact that, like vineyards and unlike the standard persistence algorithm, the \( R = DV \) decomposition is permuted with every \( O(m) \) operation. Indeed, as shown with the
natural images example in section 4, there are often more permutation operations being applied than there are column reductions. In the more standard compressed sparse matrix representations (i.e., CSC, CSR, etc.), permuting both the rows and columns generally takes at most \( O(Z) \) time, where \( Z \) is the number of non-zero entries. If the particular filtration has many cycles, permuting these compressed representations can be quite expensive, bottlenecking the computation both in theory and in practice. As a result, the more complex sparse matrix representation from [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 [13] used in the reduction algorithm to the dynamic setting. Another direction to explore would be the analysis of our approach under the cohomology computation [16], 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 [17].

References

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

[2] David Cohen-Steiner, Herbert Edelsbrunner, and Dmitriy Morozov. Vines and vineyards by updating persistence in linear time. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 119–126, 2006.

[3] Woojin Kim and Facundo Mémoli. Spatiotemporal persistent homology for dynamic metric spaces. *Discrete & Computational Geometry*, pages 1–45, 2020.

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

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

[6] Henry Adams, Tegan Emerson, Michael Kirby, Rachel Neville, Chris Peterson, Patrick Shipman, Sofya Chepushtanova, Eric Hanson, Francis Motta, and Lori Ziegelmeier. Persistence images: A stable vector representation of persistent homology. *Journal of Machine Learning Research*, 18, 2017.

[7] M Ulmer, Lori Ziegelmeier, and Chad M Topaz. A topological approach to selecting models of biological experiments. *PloS one*, 14(3):e0213679, 2019.

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

[9] Nina Otter, Mason A Porter, Ulrike Tillmann, Peter Grindrod, and Heather A Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6:1–38, 2017.

[10] Michael Lesnick and Matthew Wright. Interactive visualization of 2-d persistence modules. *arXiv preprint arXiv:1512.00180*, 2015.

[11] The RIVET Developers. Rivet, 2020.

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

[13] Chao Chen and Michael Kerber. Persistent homology computation with a twist. In *Proceedings 27th European Workshop on Computational Geometry*, volume 11, pages 197–200, 2011.

[14] Chad M Topaz, Lori Ziegelmeier, and Tom Halverson. Topological data analysis of biological aggregation models. *PloS one*, 10(5):e0126383, 2015.

[15] Lu Xian, Henry Adams, Chad M Topaz, and Lori Ziegelmeier. Capturing dynamics of time-varying data via topology. *arXiv preprint arXiv:2010.05780*, 2020.

[16] Vin De Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson. Dualities in persistent (co) homology. *Inverse Problems*, 27(12):124003, 2011.

[17] Ulrich Bauer. Ripser: efficient computation of Vietoris–Rips persistence barcodeS. *Journal of Applied and Computational Topology*, pages 1–33, 2021.

[18] Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. Phat–persistent homology algorithms toolbox. *Journal of symbolic computation*, 78:76–90, 2017.
[19] Dominique Attali, Marc Glisse, Samuel Hornus, Francis Lazarus, and Dmitriy Morozov. Persistence-sensitive simplication of functions on surfaces in linear time. In TopoInVis’09, 2009.

[20] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In Proceedings 41st annual symposium on foundations of computer science, pages 454–463. IEEE, 2000.

[21] Cecil Jose A Delfinado and Herbert Edelsbrunner. An incremental algorithm for betti numbers of simplicial complexes on the 3-sphere. Computer Aided Geometric Design, 12(7):771–784, 1995.

[22] Chao Chen and Michael Kerber. An output-sensitive algorithm for persistent homology. Computational Geometry, 46(4):435–447, 2013.

[23] Patrick Oesterling, Christian Heine, Gunther H Weber, Dmitriy Morozov, and Gerik Scheuermann. Computing and visualizing time-varying merge trees for high-dimensional data. In Topological Methods in Data Analysis and Visualization, pages 87–101. Springer, 2015.

[24] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms, pages 1131–1142. SIAM, 2013.

[25] Jean-Daniel Boissonnat and Jack Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. Computational Geometry, 16(1):35–52, 2000.

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

[27] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pages 39–48. IEEE, 2000.

[28] Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. Information Processing Letters, 76(1-2):7–11, 2000.

[29] S Kiran Kumar and C Pandu Rangan. A linear space algorithm for the lcs problem. Acta Informatica, 24(3):353–362, 1987.

[30] Jinho Baik, Percy Deift, and Kurt Johansson. 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.

[31] Jean-Daniel Boissonnat and Karthik CS. An efficient representation for filtrations of simplicial complexes. ACM Transactions on Algorithms (TALG), 14(4):1–21, 2018.

[32] Persi Diaconis and Ronald L Graham. Spearman’s footrule as a measure of disarray. Journal of the Royal Statistical Society: Series B (Methodological), 39(2):262–268, 1977.

[33] Therese Biedl, Franz J Brandenburg, and Xiaotie Deng. On the complexity of crossings in permutations. Discrete Mathematics, 309(7):1813–1823, 2009.

[34] Michael Phillip Lesnick. Multidimensional interleavings and applications to topological inference. Stanford University, 2012.

[35] Mickaël Buchet, Frédéric Chazal, Tamal K Dey, Fengtao Fan, Steve Y Oudot, and Yusu Wang. Topological analysis of scalar fields with outliers. In 31st International Symposium on Computational Geometry, pages 827–841. Schloss Dagstuhl, Leibniz-Zentrum für Informatik GmbH, 2015.

[36] Ulderico Fugacci and Michael Kerber. Chunk reduction for multi-parameter persistent homology. In 35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA, pages 37–1, 2019.

[37] Gunnar Carlsson, Gurjeet Singh, and Afra J Zomorodian. Computing multidimensional persistence. Journal of Computational Geometry, 1(1):72–100, 2010.

[38] Claudia Landi. The rank invariant stability via interleavings. arXiv preprint arXiv:1412.3374, 2014.

[39] Gunnar Carlsson and Afra Zomorodian. The theory of multidimensional persistence. Discrete & Computational Geometry, 42(1):71–93, 2009.

[40] Michael Lesnick and Matthew Wright. Computing minimal presentations and bigraded betti numbers of 2-parameter persistent homology. arXiv preprint arXiv:1902.05708, 2019.

[41] Jean-Daniel Boissonnat and Franco P Preparata. Robust plane sweep for intersecting segments. SIAM Journal on Computing, 29(5):1401–1421, 2000.
[42] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[43] Ann B Lee, Kim S Pedersen, and David Mumford. The nonlinear statistics of high-contrast patches in natural images. *International Journal of Computer Vision*, 54(1):83–103, 2003.

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

[45] Gunnar Carlsson, Tigran Ishkhanov, Vin De Silva, and Afra Zomorodian. On the local behavior of spaces of natural images. *International journal of computer vision*, 76(1):1–12, 2008.

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

[47] Jose A Perea. A brief history of persistence. *arXiv preprint arXiv:1809.03624*, 2018.

A

**Vineyard Algorithm**

**Algorithm 6** Vineyards: Tranposition Framework

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

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

1: function **TRANSPOSE**($R, V, i$)
2: 
3: \hspace{1em} pos ← columns satisfying $col_R = 0$
4: \hspace{1em} if $\text{pos}[i]$ and $\text{pos}[i + 1]$ then
5: \hspace{2em} if $V[i, i + 1] \neq 0$ then
6: \hspace{3em} $col_V(i + 1) += col_V(i)$
7: \hspace{2em} if $\exists k, l$ s.t. $\text{low}_R(k) = i, \text{low}_R(l) = i + 1$ and $R[i, l] \neq 0$ then $\triangleright O(m)$
8: \hspace{3em} if $k < l$ then
9: \hspace{4em} return $(R, V) ← (PRS_{i+1}^k P, PVS_{i+1}^k P)$
10: \hspace{2em} else
11: \hspace{3em} return $(R, V) ← (PRPS_{i+1}^k P, PVS_{i+1}^k P)$
12: \hspace{1em} else if $!\text{pos}[i]$ and $!\text{pos}[i + 1]$ then
13: \hspace{2em} if $V[i, i + 1] \neq 0$ then $\triangleright O(m)$
14: \hspace{3em} if $\text{low}_R(i) < \text{low}_R(i + 1)$ then
15: \hspace{4em} return $(R, V) ← (PRS_i^i+1 P, PV S_i^i+1 P)$
16: \hspace{2em} else
17: \hspace{3em} return $(R, V) ← (PRS_i^i+1 P S_i^i+1, PV S_i^i+1 P S_i^i+1)$
18: \hspace{1em} else if $!\text{pos}[i]$ and $\text{pos}[i + 1]$ then
19: \hspace{2em} if $V[i, i + 1] \neq 0$ then $\triangleright O(m)$
20: \hspace{3em} return $(R, V) ← (PRS_i^i+1 P S_i^i+1, PV S_i^i+1 P S_i^i+1)$
21: \hspace{2em} else if $\text{pos}[i]$ and $!\text{pos}[i + 1]$ then
22: \hspace{3em} $col_V(i + 1) += col_V(i)$
23: \hspace{2em} if $V[i, i + 1] \neq 0$ then $\triangleright O(m)$
24: \hspace{3em} return $(R, V) ← (PRP, PV P)$

**Transposition complexity details**

Consider Algorithm 6. The matrices $R, V$ are both $(m \times m)$, where $m$ is the number of simplices in the underlying filtration. Each transposition $Tr(i, i + 1)$ requires at most 2 column operations (denoted with $S_j^i$ in the algorithm) in both $R$ and $V$. A constant number of queries to the matrix representation, and exactly 1 row and column exchange. Column operations take $O(m)$ time. Querying to determine whether an entry at position $(i, i + 1)$ is non-zero depends
on the sparse matrix implementation, but is generally either $O(\log m)$ or $O(m)$. In compressed sparse matrix representation, column and row exchanges take upwards of $O(z)$ time, where $z$ is the number of non-zero entries in the matrix, however this can be made $O(1)$ time by using the data structure described in [2]. This data structure requires storing an index using at most $O(m \log m)$ bits with each non-zero entry, as well as several $O(m)$-sized auxiliary arrays to protect the matrices from row exchanges.