Alcom-FT Technical Report Series
ALCOMFT-TR-01-146

Permutation Editing and Matching via Embeddings

Graham Cormode[1], S. Muthukrishnan[2], and Süleyman Cenk Ṣahinalp[3]

[1] University of Warwick, Coventry, UK; grahamc@dcs.warwick.ac.uk
[2] AT&T Research, Florham Park, NJ, USA; muthu@research.att.com
[3] EECS, Case Western Reserve University, Cleveland, OH; cenk@cwru.edu

Abstract. If the genetic maps of two species are modelled as permutations of (homologous) genes, the number of chromosomal rearrangements in the form of deletions, block moves, inversions etc. to transform one such permutation to another can be used as a measure of their evolutionary distance. Motivated by such scenarios, we study problems of computing distances between permutations as well as matching permutations in sequences, and finding most similar permutation from a collection ("nearest neighbor").

We adopt a general approach: embed permutation distances of relevance into well-known vector spaces in an approximately distance-preserving manner, and solve the resulting problems on the well-known spaces. Our results are as follows:

- We present the first known approximately distance preserving embeddings of these permutation distances into well-known spaces.
- Using these embeddings, we obtain several results, including the first known efficient solution for approximately solving nearest neighbor problems with permutations and the first known algorithms for finding permutation distances in the "data stream" model.
- We consider a novel class of problems called *permutation matching* problems which are similar to string matching problems, except that the pattern is a permutation (rather than a string) and present linear or near-linear time algorithms for approximately solving permutation matching problems; in contrast, the corresponding string problems take significantly longer.

1 Introduction

As the first phase of the Human Genome Project approaches completion, the attention is shifting from raw sequence data to genetic maps. Comparative studies of gene loci among closely related species provide clues towards understanding the complex phylogenetic relationships between species and their evolutionary order. Genetic maps of two species can be thought of as permutations of homologous genes and the number of chromosomal rearrangements in the form of deletions, copies, inversions, transpositions to transform one such permutation to another can be used as a measure of their evolutionary distance. Computational methods for measuring genetic distance between species is an active area of research in computational genomics, especially in the context of comparative mapping [13], eg., using reversal distance [1, 4, 11], transposition distance [2, 8] or other measures. In a more general setting it is of interest to not only compute the distances between two permutations but also to find the closest gene permutation to a given one in a database or to approximately find a given permutation of genes in a larger sequence, etc. Given the representation as permutations, these can all be abstracted as permutation editing and matching problems.[1]

Permutations are ordered sequences over some alphabet with no repetitions allowed.[2] Thus, any permutation is a string, although strings are not generally permutations since they are allowed to repeat symbols. Suitable edit operations on permutations include reversals; transpositions; alphabet edits such as inserts and deletes; and symbol moves (formal definition of these operations follows). We study problems of computing pairwise edit distances, similarity searching, matching and so on, motivated by Computational Biology and other scenarios. We adopt a general approach to solving all such problems on permutations: develop an embedding of permutations into vector spaces such that the distance between the resulting vectors approximates the distance between any two permutations. Thus permutation editing and matching problems reduce to natural problems on vector spaces.

Even though we have motivated permutation editing, matching and similarity searching problems from Computational Biology applications, there are other reasons for their study. Permutations form an interesting class of combinatorial objects by themselves, and therefore it is quite natural to study the complexity of computing edit distances between permutations, and to do similarity searching. In addition, they arise in many applications. Since permutations are special cases of strings, permutation editing and matching problems give insight into the complexity of string editing and matching problems many of which are classical and still open. This will be clarified later using our results. In what follows, we will first describe the edit distance problems with permutations before describing our results.

1.1 Notation

A permutation is a sequence of symbols such that within a permutation each symbol is unique. We shall often represent these symbols as integers drawn from some range, so 1 3 2 4 is a valid permutation, but 1 2 3 2 is not. Signed permutations are permutations where each symbol can take two forms: positive and negative, eg $1^+\ 3^+\ 2^-\ 4^+$. Operations can also change the signs of symbols, and two signed permutations are considered identical only if every symbol and every sign agree. In what follows, P, Q will represent permutations, and $i, j, k \ldots$ will

[1] More complex notions of genetic distance which take into account that (1) the genome is composed of multiple chromosomes [6, 14], or (2) exact order of the genes within a genome is not necessarily known [7] have recently been proposed.

[2] Sometimes it matters in which orientation a gene occurs, and so use is made of *signed permutations*

be integers. The i'th symbol of a permutation P will be denoted as $P[i]$, and the inverse of the permutation P^{-1} is defined so that if $P[i] = j$ then $P^{-1}[j] = i$. We can also compose one permutation with another, so $(P \circ Q)[i] = P[Q[i]]$. The "identity permutation" is the permutation of for which $P[i] = i$ for all i. For uniformity, we shall extend *all* permutations P by adding $P[0] = 0$ and $P[n+1] = n+1$, where n is the length of P. This allows the first and last symbols of P to be treated identically to the other symbols. All logarithms will be taken to base 2, and rounded up, so $\log n$ should be interpreted as $\lceil \log_2 n \rceil$.

1.2 Permutation Editing and Matching Problems

First we focus on defining distances between permutations. Consider any two permutations P and Q over some alphabet set. The following distances are of interest:

Reversal Distance: Denoted $r(P, Q)$, reversal distance is defined as the minimum number of reversals of contiguous subsequences necessary to transform permutation P into Q. So if P is a permutation $P[1] \ldots P[n]$, then a Reversal operation with parameters i, j $(i < j)$ results in the permutation $P[1] \ldots P[i-1], P[j], P[j-1] \ldots P[i+1], P[i], P[j+1] \ldots P[n]$. If P is a signed permutation then additionally the sign of each symbol $P[j] \ldots P[i]$ is switched (from plus to minus and vice-versa). This distance has been well-studied, and is shown to be NP-hard to find exactly [3]. The best approximation algorithm for this distance is a 3/2 factor algorithm due to Christie [4].

Transposition Distance: Denoted $t(P, Q)$, transposition distance is defined as the minimum number of moves of contiguous subsequences to arbitrary new locations necessary to transform permutation P into Q. Bafna and Pevzner [2] give a 3/2 approximation algorithm for transposition distance. Given $P[1] \ldots P[n]$, a transposition with parameters i, j, k $(i < j < k)$ gives
$P[1] \ldots P[i-1], P[j], P[j+1] \ldots P[k], P[i], P[i+1] \ldots P[j-1], P[k+1] \ldots P[n]$.

Permutation Edit Distance: The permutation edit distance between two permutations, $d(P, Q)$ is the minimum number of moves required to transform P into Q. A move can take a single symbol and place it at an arbitrary new position in the permutation. Hence a move with parameters i, j $(i < j)$ turns $P[1] \ldots P[n]$ into $P[1] \ldots P[i-1], P[i+1] \ldots P[j], P[i], P[j+1] \ldots P[n]$. This distance is analogous to the Levenshtein edit distance on strings, since in both cases an optimal set of edit operations will isolate a longest common subsequence and leave this unaltered, while performing edit operations on every other symbol.

Symbol Indels: Each of the above distances can be augmented by additionally allowing insertions and deletions of a single symbol at a time. This takes care of the fact that the alphabet set in two permutations need not be identical.

It will be of interest to (1) combine all operations (transposition, reversal, symbol moves) and define the cumulative distance between any two permutations involving minimum number of operations, and (2) generalize the definitions so that at most one of P or Q is a string (as opposed to a permutation). If both P and Q are strings, we are in the familiar territory of string matching.

1.3 Our Results

Our main results are threefold. We give them in outline; the precise bounds are given in later sections.

In Section 2 we present embeddings of permutation distances into well-understood spaces such as Hamming or Set Intersection. The embeddings preserve the original distances within a small constant or logarithmic factor, and are small polynomial in size. These are the first such approximately distance-preserving embeddings in the literature for permutation distances. The embeddings use a technique we develop in this paper of capturing the relative layout of pairs of symbols in the permutations by two dimensional matrices. Our embeddings capture the relevant pairs that help approximate the permutation distances accurately and the resulting matrices are often sparse. We believe that this approach to embedding distances will be of independent interest.

The embeddings above immediately give approximation algorithms for computing the distance between two permutations in (near) linear time. In addition, we use the embeddings above to solve several other algorithmic problems of which we list the following two as important examples: (1) Computing permutation distances in a distributed or Communication Complexity setting wherein the number of bits exchanged is the prime criterion for efficiency, and also in a "data streaming" model wherein data is scanned in order as it streams by and only small amount of space is allotted for storage. Streaming algorithms are known for vector distance computations; nothing was known beforehand for permutations — moreover, no streaming algorithms were known for any string distances. (2) Providing efficient approximate nearest neighbor searches for permutation distances. We provide the first known approximate algorithms for these problems that avoid the dimensionality bottleneck. These are all described in Section 3.

The problem of *Approximate Permutation Matching* is, given a long text string and a pattern permutation, to find all occurrences of the pattern in the text with at most a given threshold of distance. This is the generalization of the standard k-mismatches problem with strings (find all text locations wherein the pattern occurs with at most k mismatches) to other edit distances, and a restriction since the pattern is required to be a permutation. In Section 4 we present highly efficient, linear or near-linear time approximations for the permutation matching problems. This is intriguing since approximately solving string matching problems with corresponding distances seems to be harder, with best known algorithms taking much longer. For example, approximating string matching with edits takes time $\Omega(nm)$ for n-long text and m-long pattern where edits are transpositions, character indels and substitutions, and at least $\Omega(n \log^3 m)$ even if only substitutions are allowed [10]! In contrast, our algorithms take only $O(n+m)$ or $O(n \log m)$ time for permutations.

Our embeddings give other results such as efficient clustering algorithms for permutations, and other similarity problems. We do not discuss them further here since they follow in a straightforward way by combining our embeddings with results known for the target spaces such as Hamming, L_1, and Set Intersection. This is a welcome side-effect of our use of embeddings.

2 Embeddings of Permutation Distances

2.1 Reversal Distance

For signed permutations, we replace every positive element i^+ with the pair i' i'' and every i^- with i'' i'. The reversal distance of two unsigned versions is the same as the reversal distance of the original permutations. We define a two dimensional matrix, $R(P)$, as a binary matrix of size $(n+2) \times (n+2)$. For all $0 \leq j < i \leq n+1$, set $R(P)[i,j]$ to 1 if $i > j$ and i is adjacent to j in P, that is, if either $P^{-1}[i] = 1 + P^{-1}[j]$ or $P^{-1}[j] = 1 + P^{-1}[i]$. Otherwise, $R[i,j] = 0$. We set $R[i,i] = 0$ for all i, and the matrix is only populated above this main diagonal. Recall that the reversal distance between two permutations is denoted as $r(P,Q)$. The Hamming distance between two bit vectors X and Y is denoted $H(X,Y)$. The Hamming distance between two matrices is the Hamming distance between two vectors obtained by linearizing the two matrices in any manner.

Theorem 1. $r(P,Q) \leq \frac{1}{2}H(R(P),R(Q)) \leq 2r(P,Q)$

Proof. We extend the notion of Reversal Breakpoints given in Section 2 of [11] which is defined on a single permutation. Define a *Reversal Breakpoint of* P *relative to* Q as a location, i, where the symbol following $P[i]$ in P is not adjacent to $P[i]$ where it occurs in Q. Formally, this is when $|Q^{-1}[P[i]] - Q^{-1}[P[i+1]]| \neq 1$. We denote the total number of such breakpoints as $\phi(P,Q)$. Clearly, if $P = Q$, then $\phi(P,Q) = 0$, and this is the only way in which the count is zero. In transforming P into Q using reversals, our goal is to reduce ϕ to zero. A reversal affects two locations, so we can reduce ϕ by at most two per move, which gives a lower bound. It is also the case that we can always convert P into Q using at most $\phi(P,Q)$ reversals. This follows from considering relabelling Q as the identity permutation, and applying this same relabelling to P generating $Q^{-1} \circ P$. The reversal breakpoints of P relative to Q then become precisely the reversal breakpoints of $Q^{-1} \circ P$ relative to the identity permutation, and consequently, the permutation can be edited using at most this number of reversals, following from Theorem 1 in [11]. Hence $r(P,Q) \leq \phi(P,Q) \leq 2r(P,Q)$.

It remains to show that $H(R(P),R(Q)) = 2\phi(P,Q)$. Suppose that $R(P)[i,j] = 1$ and $R(Q)[i,j] = 0$. This means that i and j are adjacent in P but not in Q. If we sum the number of distinct pairs i,j which are adjacent in P but not in Q, then this finds $\phi(P,Q)$. This is because every breakpoint will generate such a pair, and such pairs can only arise from breakpoints. An identical argument follows when $R(P)[i,j] = 0$ and $R(Q)[i,j] = 1$, yielding $\phi(Q,P)$. Since $\phi(Q,P) = \phi(P,Q)$, it follows that $H(R(P),R(Q))$ counts each breakpoint exactly two times.

2.2 Transposition Distance

We define $T(P)$, a binary matrix for a permutation P such that $T(P)[i,j] = 1$ if j immediately follows i in P, ie if $P^{-1}[i] + 1 = P^{-1}[j]$.

Theorem 2. $t(P,Q) \leq \frac{1}{2}H(T(P),T(Q)) \leq 3t(P,Q)$

Proof. Define a *Transposition Breakpoint* in a permutation P relative to another permutation Q as a location, i, such that $P[i+1]$ does not immediately follow $P[i]$ when it occurs Q, [3] that is $Q^{-1}[P[i]]+1 \neq Q^{-1}[P[i+1]]$. Let the total number of such transposition breakpoints between P and Q be denoted as $tb(P,Q)$. Observe that to convert P to Q we must remove all breakpoints, since $tb(Q,Q) = 0$. A single transposition affects three locations and so could 'fix' at most three breakpoints — this gives the lower bound. Also, we can always fix at least one breakpoint per transposition using the trivial greedy algorithm, which gives the upper bound. Hence $t(P,Q) \leq tb(P,Q) \leq 3t(P,Q)$.

We now need to show that $H(T(P),T(Q)) = 2tb(P,Q)$: clearly, $tb(P,Q) = tb(Q,P)$. $T(P)[i,j] = 1$ and $T(Q)[i,j] = 0$ if and only if there is a transposition breakpoint in Q at the location of i, so summing these contributions generates $tb(P,Q)$. A symmetrical argument holds when $T(P)[i,j] = 0$ and $T(Q)[i,j] = 1$, and these two cases summed generate exactly $H(T(P),T(Q)) = 2tb(P,Q)$.

2.3 Permutation Edit Distance

We show how to embed Permutation Edit Distance into Set Intersection Size up to a factor of $\log n$. We shall define $A(P)$ as an $n \times n$ binary matrix derived from a permutation of length n, P. $A_k(P)[i,j]$ is set to one if symbol i occurs a distance of exactly 2^k before j in P. Otherwise, $A_k(P)[i,j] = 0$. $A(P)$ is formed by taking the union of the matrices $A_0 \ldots A_{\log n-1}$. That is, $A(P)[i,j] = 1 \iff \exists k.\ (P^{-1}[i]+2^k = P^{-1}[j])$. Note that $A(P)$ is a binary matrix, and $n \log n + \Theta(n)$ entries are 1.

Also, let $B(Q)$ be an $n \times n$ binary matrix defined on a permutation Q such that $B(Q)[i,j]$ is zero if i occurs before j in Q. Otherwise $B(Q)[i,j] = 1$. Thus, $B(Q)[i,j] = 0 \iff (Q^{-1}[i] < Q^{-1}[j])$ In this matrix, $n^2/2 + \Theta(n)$ entries are 1. Finally, define $D(P,Q)$ as the size of the intersection between $A(P)$ and $B(Q)$. Put another way, this intersection can be calculated using multiplication of the elements of the matrices, pairwise: $D(P,Q) = \sum_{i,j}(A(P)[i,j] \times B(Q)[i,j])$.

Theorem 3. $d(P,Q) \leq D(P,Q) \leq \log n \cdot d(P,Q)$.

Proof. i) $D(P,Q) \leq \log n \cdot d(P,Q)$
Consider the pairs (i,j) such that $A(P)[i,j] = B(Q)[i,j] = 1$. The number of such pairs is exactly $D(P,Q)$. Each of these pairs has i occurring before j in P, but the other way round in Q, and so one of either i or j must be moved to turn P into Q. So in effect, these pairs represent a "to-do" list of changes that must be made. By construction of A, any symbol i appears at most $\log n$ times amongst these pairs. Hence whenever a move is made, at most $\log n$ pairs can be removed from this to-do list. It therefore follows that in each move, D can change by at most $\log n$. If at every step we change D by at most $\log n$, then this bounds the minimum number of operations possible to transform P into Q as $D(P,Q)/\log n \leq d(P,Q)$

[3] As usual, we extend all permutations so that the first symbol is 0 and their last is $n+1$.

ii) $d(P,Q) \leq D(P,Q)$
We shall show the bound by concentrating on the fact that an optimal edit sequence preserves a Longest Common Subsequence of the two sequences. Note that an optimal edit sequence will have length $n - LCS(P,Q)$: every symbol that is not moved must form part of a common subsequence of P and Q and so an optimal edit scheme will ensure that this common subsequence is as long as possible. Consider the relabelling of Q so that for all i, $Q[i]$ is relabelled with i. We analyze the effect of applying this relabelling to P and examine its longest increasing subsequence. Call this relabelled sequence P'. Clearly, the longest common subsequence of P and Q is not altered, since we have just relabelled distinct symbols. Because Q is replaced by a strictly increasing sequence, it follows that each Longest Common Subsequence of P and Q corresponds exactly to one Longest Increasing Subsequence of P', whose length is denoted by $LIS(P')$. Qualitatively, what D told us was that we count 1 if symbol is 2^k to the right of the i'th location in P but is anywhere to the left in Q. When we relabel according to Q, this translates so we count 1 if symbol i is greater than an symbol 2^k to its right.

We shall split P' into two subsequences, one which consists only of the symbols at odd locations in P', and the other of the symbols which occur at even locations. Symbols of P' will now be referred to as 'odd symbols' or 'even symbols': this refers only to their location, not whether the value of an symbol is odd or even. Suppose s_{odd} is the length of a longest increasing subsequence of symbols at odd locations in P', and s_{even} is similarly defined for the even symbols. Define $b(P')$ as the number of locations ('sequence breakpoints') where $P'[i] > P'[i+1]$

Lemma 1. *$LIS(P') \geq s_{odd} + s_{even} - b(P')$.*

Proof. Let S_{even} represent an increasing sequence of even symbols whose length is s_{even}, and define S_{odd} similarly. We shall see how we can build a longer increasing subsequence starting from each of the subsequences of even and odd symbols. Consider an symbol of S_{even}, $P'[i]$ and the subsequent symbol of S_{even}, $P'[j]$. There is at least one odd symbol separating these two symbols when they occur in P'. Now, either all odd symbols that occur at locations between i and j have values between $P'[i]$ and $P'[j]$, in which case we could extend the increasing sequence S_{even} by including these symbols; or else they are all less than $P'[i]$ or greater than $P'[j]$. In either case, then there is a contribution of at least one to $b(P')$ from these intervening symbols. This allows us to conclude that from the increasing sequence S_{even}, then we can form an increasing sequence of length at least $2s_{even} - b(P')$, as there are $s_{even} - 1$ consecutive pairs of symbols from S_{even}, and in addition we can also consider the sequence before the first symbol. Similarly, from S_{odd}, we can find an increasing sequence of length at least $2s_{odd} - 1 - b(P')$. Further, depending on whether $|P'|$ is odd or even, we can always increase one of these bounds by 1, by considering the effect of the last member of S_{odd} and the subsequent even symbols if $|P'|$ is even, or the effect with the last of S_{even} and subsequent odd symbols if $|P'|$ is odd. We know that each of these generated increasing sequences of P' is of length at most $LIS(P')$ by definition of $LIS(P')$. Summing these, we find that $2s_{odd} + 2s_{even} - 2b(P') \leq 2LIS(P')$.

If we consider what $b(P')$ represents, we compare every $P'[i]$ to $P'[i+1]$ and count one for every disordered pair. This is telling us that the considered pair of symbols occur in P in the opposite order to which they occur in Q, by construction of P'. So $b(P')$ is exactly equivalent to the contribution to $D(P, Q)$ from $A_0 \cap B$. If we now split and consider P'_{odd} (and P'_{even}), the subsequences of P' formed by taking all the symbols at odd (even) locations, we note that these have exactly the same structure, and have only the self-contained comparisons of $A_1 \cap B$, $A_2 \cap B$ to $A_{\log n-1} \cap B$. We can carry on splitting each sequence recursively into odd and even sequences, until we can split no further. At the last level, all that remains are $|P'| = |P|$ single symbols, which each constitute a trivial increasing subsequence of length one. Telescoping the inequality, we find that $LIS(P') \geq |P| - b(P') - b(P'_{even}) - b(P'_{odd}) - b(P'_{odd_{odd}}) - b(P'_{odd_{even}}) \ldots$ If we sum all these b's, we get exactly $D(P, Q)$. Hence we conclude that $LCS(P, Q) = LIS(P') \geq |P| - D(P, Q)$. Rearranging and substituting, we find $D(P, Q) \geq n - LCS(P, Q) = d(P, Q)$, as required.

2.4 Combining All Operations

We consider the compound distance allowing the combination of reversals, transpositions and permutation editing (moving a single symbol). Denote this distance as $\tau(P, Q)$. We make use of the transformation R from Section 2.1, and omit the simple proof.

Theorem 4. $\tau(P, Q) \leq \frac{1}{2}H(R(P), R(Q)) \leq 3\tau(P, Q)$.

These embedding techniques can also be adapted for a large range of permutation distances. Embeddings can be obtained for variations where inserts and deletes are permitted for any of the distances already described; when one of the sequences is allowed to be a string rather than a permutation; and in the case of signed permutations. Exact details of these embeddings are omitted for brevity.

3 Implications of the Embeddings

We can immediately find algorithmic applications of our embeddings. On the whole, these rely on known results for the Hamming space.

Approximating Pairwise Distances. The embeddings allow distance approximations to be made efficiently in a communication setting. We have the following scenario: there are two communicants, A and B, who each hold a permutation P and Q respectively, and they wish to communicate in such a way to calculate the approximate distance between their permutations.

Theorem 5. *There is a single round communication protocol to allow reversal (transposition) distance approximation up to a factor of $2+\epsilon$ (respectively $3+\epsilon$) with a message of size $O(\log n \log(1/\delta)/\epsilon^2)$. The protocol succeeds with probability $1-\delta$.*

This follows from known communication results on Hamming distance such as Corollary B of [5]. Now suppose that we have a number of permutations, and we wish to be able to rapidly find the distance between any pair of them. Traditional methods would suggest that for each pair, we should take one and relabel it as the identity permutation, and then solve the sorting by reversals or sorting by transpositions problem for the correspondingly relabelled permutation. We claim that, given a near linear amount of preprocessing, this problem can be solved exponentially faster.

Corollary 1. *With a linear amount of preprocessing, the Reversal distance (respectively, Transposition distance) between any pair of permutations of length n can be approximated in time $O(\log n \log(1/\delta)/\epsilon^2)$ up to a factor of $2+\epsilon$ (resp. $3+\epsilon$).*

This follows from the above statement, since whenever we have a one round communication protocol, we can precompute and store the message that would be sent for each permutation. Pairwise approximation can then be carried out by comparing the two corresponding precomputed messages, which requires time linear in the size of the message.

Approximate Nearest Neighbors. The problem is to preprocess a collection of permutations so that given a new query permutation, the closest permutation from the collection can be found. This problem is analogous to vector nearest neighbors under Hamming metric [9, 12]. The crux here is to avoid the dimensionality curse: that is, design a polynomial space data structure that answers queries in time polynomial in the query and sublinear in the collection size.

Theorem 6. *We can find approximate nearest neighbors under Reversal distance (respectively Transposition distance and compound distances thereof) up to a factor of $2+\epsilon$ (respectively $3+\epsilon$) with query time $O(\ell \cdot n^{1/(1+\epsilon)})$, where n is the number of sequences in the database, and ℓ the size of the universe from which sequence symbols are drawn.*

Proof. This follows immediately from the results for Approximate Nearest Neighbors in [9] and [12]. Some care is needed, since for efficiency we need to ensure that the sampling at the root of the Locality-Sensitive Hash functions used therein does not attempt to sample directly from the quadratic ($O(\ell^2)$) space of the matrices of the embeddings. Instead, we consider in turn each adjacent pair in a permutation, and use hash functions to determine whether this pair would have been picked by the sampling scheme.

Distance Estimation in the Streaming Model An additional feature of the embeddings is that they lead themselves to solving problems in the streaming model.

Theorem 7. *If the sequences arrive as arbitrarily interleaved streams, approximations for the Transposition distance or Reversal distance can be computed using storage of size* $O(\log \ell \log(1/\delta)/\epsilon^2)$ *such that the Reversal distance (respectively Transposition distance) can be approximated to a factor of* $2 + \epsilon$ *(resp.* $3 + \epsilon$*) with probability* $1 - \delta$.

Proof. Since each non-zero entry in the transformation matrices comes from information about adjacent pairs in the permutation, we can parse the permutation as a stream of tuples, so $\ldots i, j, k \ldots$ is viewed as $\ldots (i,j),(j,k) \ldots$. The streaming algorithm of [5] can then be used on the induced bitstring (only non-zero bits need to be handled). Although the matrix space is $O(\ell^2)$, the space needed will still be $O(\log \ell)$ in size, and can be computed with a linear pass over each permutation.

4 Approximate Permutation Matching

The counting version of Approximate Permutation Matching is stated as follows: Given a text string T of length n, and a pattern permutation P of length m, find the approximate cost of aligning the pattern against each location in the text. That is, for each i find the appropriate distance $d[i]$ between $T[i : i+m-1]$ and $P[1 : m]$. Naively using the transformations of our distances would be expensive; we take advantage of the fact that because the embeddings are based on pairwise comparisons, the approximate cost can be calculated incrementally with only a small amount of work.

Theorem 8. *i) Approximate permutation matching for reversal distance can be solved in time* $O(n+m)$*; each* $d[i]$ *is approximated to a factor of 2.*

ii) Approximate permutation matching for transposition distance can be solved in time $O(n+m)$*; each* $d[i]$ *is approximated to a factor of* 3.

Proof. We must allow insertions and deletions to our sequences since in this scenario we cannot insist that we will always find exact permutations at each alignment location. We shall make use of extended embeddings which allow these approximations to be made. It is important to note that although these embeddings are described in terms of quadratic sized matrices, we do not construct these matrices, but instead concentrate only on the linear number of non-zero entries in these figurative matrices. We shall prove both claims together, since we take advantage of common properties of the embeddings.

Suppose we know the cost of aligning $T[i \ldots i+m-1]$ against P, and we now want to find the cost for $T[i+1 \ldots i+m]$. This is equivalent to adding a character to the end of $T[i \ldots i+m-1]$ and removing one from the front. So only two adjacencies are affected — at the start and end of the subsequence. This affects only a constant number of symbols in our matrices. Consequently, we need only perform a constant amount of work to update our count of the number of transposition or reversal breakpoints, provided we have precomputed the inverse of the pattern P in time $O(m)$. To begin the process, we imagine aligning P with a location to the left of T so that there is no overlap of pattern and text. Initially, the distance between P and $T[-m \ldots 0]$ is defined as m. From this position, we can advance the pattern by one location at a time and do a constant amount of work to update the count. The total time required is $O(n + m)$.

Theorem 9. *Approximate Permutation Matching can be solved for Permutation Edit Distance in time $O(n \log m)$.*

Proof. We make use of the transformation for permutation edit distance, and so our result will be accurate up to a factor of $\log m$. We can use the trick of relabelling P as $1\ 2 \ldots m$, and relabelling T accordingly as we go along. Suppose we have found the cost of matching $T[i \ldots i+m-1]$ against P. We can advance this match by one location to the right by comparing $T[i+m]$ with the $\log m$ locations $T[i+m-1], T[i+m-2], T[i+m-4] \ldots$. Each pair of the form $T[i+m-2^k] > T[i+m]$ that we find adds one to our total. At the same time, we maintain a record of the L_1 difference between the number of symbols in P missing from $T[i \ldots i+m-1]$ (since each of these must participate in an insertion operation to transform $T[i \ldots i+m-1]$ into P). This can be updated in constant time using $O(P)$ space. We can step the left end of a match by one symbol in constant time if we also keep a record for each $T[i]$ how many comparisons it caused to fail from symbols to the right — we reduce the count by this much to find the cost of $T[i+1 \ldots i+m]$ from $T[i \ldots i+m]$. In total we do $O(\log m)$ work per step, giving a total running time of $O(n \log m)$.

5 Discussion

We present the first known results embedding various permutation distances into Hamming and Set Intersection spaces in an approximately distance preserving manner. These embeddings are approximate, since finding the distances exactly is provably hard. From these embeddings, a wide variety of problems can be solved, the full extent of which is beyond the scope of this paper. In particular, we have described how the embeddings enable the solution of traditional problems such as pair-wise distance estimation and nearest neighbors; and novel problems, such as approximate permutation matching and measurements in the streaming model. These results are of interest in Computational Biology as well as for foundational reasons since analogous problems are open for strings. In solving approximate permutation matching problems, we obtained linear or near-linear time approximation algorithms while their string counterparts take significantly longer. We hope that our study of permutation distances gives insights that may help solve the corresponding open problems on string distances. A candidate problem to think about seems to be approximating the longest common subsequence, a dual of our permutation edit distances.

Acknowledgements

The first author wishes to thank Mike Paterson for some fruitful discussions about this work; in particular, for suggesting the form of Lemma 1 that enabled the proof of Theorem 3. We also thank the anonymous reviewers for their comments. This work was partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

References

1. V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of the 34th Annual Symposium on Foundations of Comptuer Science*, pages 148–157, Palo Alto, CA, 1993. IEEE Computer Society Press.
2. Vineet Bafna and Pavel A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, May 1998.
3. A. Caprara. Sorting by reversals is difficult. In *Proceedings of the First International Conference on Computational Molecular Biology*, pages 75–83, 1997.
4. David A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 244–252, San Francisco, California, 25–27 January 1998.
5. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L^1-difference algorithm for massive data streams. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 501–511, 1999.
6. Vincent Ferretti, Joseph H. Nadeau, and David Sankoff. Original synteny. In *Combinatorial Pattern Matching, 7th Annual Symposium*, volume 1075 of *Lecture Notes in Computer Science*, pages 159–167. Springer, 1996.
7. Leslie Ann Goldberg, Paul W. Goldberg, Mike Paterson, Pavel Pevzner, Süleyman Cenk Ṣahinalp, and Elizabeth Sweedyk. The complexity of gene placement. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 386–395, N.Y., January 17–19 1999. ACM-SIAM.
8. Qian-Ping Gu, Shietung Peng, and Hal Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 17 January 1999.
9. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 604–613, 1998.
10. Howard Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, November 1993.
11. J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1/2):180–210, January 1995.
12. E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 614–623, 1998.
13. J. H. Nadeau and B. A. Taylor. Lengths of chromosome segments conserved since divergence of man and mouse. *Proc. Nat'l Acad. Sci. USA*, 81:814–818, 1984.
14. D. Sankoff and J. Nadeau. Conserved synteny as a measure of genomic distance. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 71, 1996.