Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement*

John Kececioglu† David Sankoff‡

October 9, 1992; revised February 19, 1993

Abstract Motivated by the problem in computational biology of reconstructing the series of chromosome inversions by which one organism evolved from another, we consider the problem of computing the shortest series of reversals that transform one permutation to another. The permutations describe the order of genes on corresponding chromosomes, and a *reversal* takes an arbitrary substring of elements and reverses their order.

For this problem we develop two algorithms: a greedy approximation algorithm that finds a solution provably close to optimal in $O(n^2)$ time and $O(n)$ space for an n element permutation, and a branch and bound exact algorithm that finds an optimal solution in $O(mL(n,n))$ time and $O(n^2)$ space, where m is the size of the branch and bound search tree and $L(n,n)$ is the time to solve a linear program of n variables and n constraints. The greedy algorithm is the first to come within a constant factor of the optimum; it guarantees a solution that uses no more than twice the minimum number of reversals. The lower and upper bounds of the branch and bound algorithm are a novel application of maximum weight matchings, shortest paths, and linear programming.

In a series of experiments we study the performance of an implementation on random permutations and permutations generated by random reversals. For permutations differing by k random reversals we find that the average upper bound on reversal distance estimates k to within one reversal for $k < \frac{1}{2}n$ and $n \leq 100$. For random permutations we find that the average difference between the upper and lower bounds is less than 3 reversals for $n \leq 50$. Due to the tightness of these bounds we can solve to optimality random permutations on 30 elements in a few minutes of computer time.

Keywords Computational biology, approximation algorithms, branch-and-bound algorithms, edit distance, permutations, chromosome inversions

---

*A version of this paper appeared in *Algorithmica* 13:1/2, 180–210, 1995.

†Department of Computer Science, The University of Georgia, Athens, GA 30602, USA. Electronic mail: `kece@cs.uga.edu`. Research supported by a postdoctoral fellowship from the Program in Mathematics and Molecular Biology at Berkeley under NSF Grant DMS–8720208, and by a fellowship from the Centre de recherches mathématiques of the Université de Montréal.

‡Centre de recherches mathématiques, Université de Montréal, Case postal 6128, succursale A, Montréal, Québec H3C 3J7. Electronic mail: `sankoff@ere.umontreal.ca`. Research supported by grants from the Natural Sciences and Engineering Research Council of Canada, and the Fonds pour la formation de chercheurs et l'aide à la recherche (Québec). DS is a fellow of the Canadian Institute for Advanced Research.

1 Introduction

Much research has been devoted to efficient algorithms for the edit distance between two strings, that is, the minimum number of insertions, deletions, and substitutions to transform one string into another. Motivation comes in large part from computational biology: at the level of individual characters, genetic sequences mutate by these operations, so edit distance is a useful measure of evolutionary distance.

At the chromosome level, however, genetic sequences mutate by more global operations such as the reversal of a substring (*inversion*), the deletion and subsequent reinsertion of a substring far from its original site (*transposition*), duplication of a substring, and the exchange of prefixes or suffixes of two sequences in the same organism (*translocation*). An inversion, which takes takes a substring of unrestricted size and replaces it by its reverse in one operation, has the effect of reversing the order of the genes contained within the substring, and is perhaps the most common of these operations [21, pages 174–175], especially in organisms with one chromosome.

For example the only major difference between the gene orders of two of the most well-known bactera, *Escherichia coli* and *Salmonella typhimurium*, is an inversion of a long substring of the chromosomal sequence [17]. In plants, Palmer, Osorio and Thompson [18] modeled the evolution of part of the pea chloroplast genome, which is also a single chromosome, in terms of five successive overlapping inversions as we illustrate in Figure 1. And in the fruit fly, genus *Drosophila*, inversions are a far more frequent reflection of differences between and within species than translocation or other processes [4, page 155].

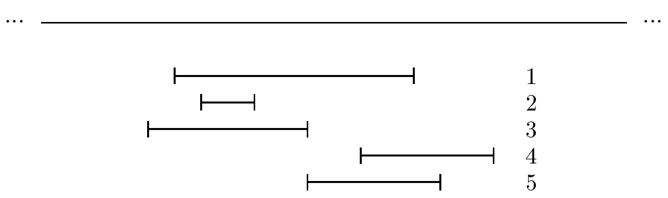


Figure 1 Evolution of the pea chloroplast genome by five overlapping inversions.

The importance of inversion in these examples suggests that algorithmic study of gene rearrangement by inversions alone is a worthwhile first step in the study of evolutionary distance at the level of the chromosome. Once this problem is understood, other processes such as transposition [19] and translocation [16] can be added to refine the model.

In the mathematical problem that we consider, we are given the order of n genes in two related single-chromosome organisms or two related organelles, which we represent by permutations $\sigma = (\sigma_1\ \sigma_2\ \cdots\ \sigma_n)$ and $\tau = (\tau_1\ \tau_2\ \cdots\ \tau_n)$.[1] In this notation σ_i denotes $\sigma(i)$. Such gene orders often come from genetic *maps* that are the distillation of the work of many experimental geneticists. In current practice, the positions of the genes are increasingly found by sequence comparison or DNA hybridization, as opposed to the mapping experiments of traditional genetics.

We model an inversion by the reversal of an interval of elements. Formally, a *reversal* of interval $[i, j]$ is the permutation[2]

$$\rho = \begin{pmatrix} i & i+1 & \cdots & j \\ j & j-1 & \cdots & i \end{pmatrix}.$$

Applying ρ to σ by the composition[3] $\sigma \circ \rho$ has the effect of reversing the order of genes σ_i, σ_{i+1}, …, σ_j. Our problem is the following.

Definition The *reversal distance problem* on permutations is, given permutations σ and τ, find a series of reversals $\rho_1, \rho_2, \ldots, \rho_d$ such that

$$\sigma \circ \rho_1 \circ \rho_2 \cdots \rho_d = \tau,$$

and d is minimum. □

We call d the *reversal distance*[4] between σ and τ. Like edit distance, it satisfies the axioms of a metric. Reversal distance measures the amount of evolution that must have taken place at the chromosome level, assuming evolution proceeded by inversions.

Notice that the reversal distance between σ and τ is equal to the reversal distance between $\tau^{-1} \circ \sigma$ and the identity permutation $\imath$, where τ^{-1} denotes the inverse of τ.[5] Hence we can take as input the permutation $\pi = \tau^{-1}\sigma$ and compute its distance from $\imath$. We call this formulation of the problem, *sorting by reversals*. Note also that any algorithm for the reversal distance between two strings that does not exploit a bounded-size alphabet must, as a special case, solve the reversal distance problem on permutations.

From an algebraic point of view, reversals generate the group of permutations under composition. Given an arbitrary group element π, we seek a shortest product of generators $\rho_1\rho_2 \cdots \rho_d$ that equals π.

Related work

Little is known about reversal distance: even its computational complexity is open. The only reference to an algorithm appears to be Watterson, Ewens, Hall, and Morgan [24], which gives the first definition of the problem and a heuristic for computing reversal

---

[1] Genes in one organism may be missing in the other. We assume however that such genes can be removed from the analysis, and that gene insertions and deletions can be analyzed separately.

[2] This notation is shorthand for $\rho(i) = j$, $\rho(i+1) = j-1$, and so on. Outside interval $[i, j]$, ρ leaves elements unchanged.

[3] The *composition* of permutations σ and ρ indicated by $\sigma \circ \rho$ is a permutation π where $\pi(i) = \sigma(\rho(i))$.

[4] We sometimes refer to d informally as the *inversion distance*.

[5] The *identity* permutation $\imath$ is $(1\ 2\ \cdots\ n)$, and the *inverse* of permutation π is the permutation π^{-1} satisfying $\pi^{-1} \circ \pi = \imath$.

distance, which we describe in Section 2. Since there are so few references, we can give a fairly exhaustive coverage of related work.

From the perspective of edit distance, Wagner [23] is interesting. Wagner considers the problem of computing the minimum number of insertions, deletions, substitutions, and transpositions of adjacent characters to convert one string to another, and shows that if the operations are restricted to deletion and transposition, the problem is NP-complete. In our problem, the permutations are effectively strings of length n without repeated characters, and we allow reversal of an arbitrary number of characters. If we restrict our problem to reversals of two characters, in other words the adjacent transpositions of Wagner, the reversal distance between σ and τ reduces to the so-called inversion number of $\pi = \tau^{-1}\sigma$, which is the number of $i < j$ such that $\pi_i > \pi_j$, a quantity clearly computable in polynomial time [13, page 11]. Tichy [22] also considers a variation of edit distance, but is less closely related to our work. More recently Schöniger and Waterman [20] present a heuristic for computing edit distance when only non-overlapping inversions are allowed.

From the perspective of sorting, related work is by Gates and Papadimitriou [9]. They consider the problem of sorting a permutation by *prefix reversals*,[6] which are reversals of the form $[1, i]$, and derive bounds on the diameter of the problem. The *diameter* of the prefix reversal problem, which we denote by $d_{\text{prefix}}(n)$, is the maximum of the minimum number of prefix reversals over all permutations on n elements. Gates and Papadimitriou show that $d_{\text{prefix}}(n) \leq \frac{5}{3}n + \frac{5}{3}$, and that for infinitely many n, $d_{\text{prefix}}(n) \geq \frac{17}{16}n$. Under the requirement that each element is reversed an even number of times, which may be appropriate if elements have an orientation (see Section 5), they show $\frac{3}{2}n - 1 \leq d_{\text{prefix}}(n) \leq 2n + 3$. In other work, Aigner and West [1] consider the diameter of sorting when the operation is reinsertion of the first element, and Amato, Blum, Irani, and Rubenfeld [2] consider a variation inspired by the problem of reversing trains on a track.

For our problem of sorting by unrestricted reversals it appears tighter bounds on the diameter are possible. The heuristic of Watterson et al. [24] sorts any n-element permutation in $n-1$ reversals, so, writing $d(n)$ for the diameter of our problem, we know $d(n) \leq n - 1$. From the other direction Gollan [10] has conjectured[7] that a particular permutation, which we denote by $\mathcal{G}_n$, requires $n-1$ reversals and has verified this for n up to 12. Recursively,

$$\mathcal{G}_{n+1} \quad = \quad \begin{cases} (1), & n \text{ is zero;} \\ \Big(\mathcal{G}_n(1) \;\; \mathcal{G}_n(2) \;\; \cdots \;\; \mathcal{G}_n(n-1) \;\; n+1 \;\; \mathcal{G}_n(n)\Big), & n \text{ is odd;} \\ \Big(\mathcal{G}_n(1) \;\; \mathcal{G}_n(2) \;\; \cdots \;\; \mathcal{G}_n(n-2) \;\; n+1 \;\; \mathcal{G}_n(n) \;\; \mathcal{G}_n(n-1)\Big), & n \text{ is even.} \end{cases}$$

Using the lower bound algorithm developed in Section 3.2 we have verified the conjecture for n up to 200 when $n \bmod 3 = 1$, and for $n \bmod 3 \neq 1$ the computation showed $\mathcal{G}_n$ requires $n - 2$ reversals. If Gollan's conjecture is true, $d(n) = n - 1$.

Note that studying the diameter of the problem, and algorithms that meet the diameter, does not imply a result concerning quality of approximation. For example,

[6] This is also known as the *pancake flipping problem*.

[7] Gollan's full conjecture is somewhat stronger: that for every n, $\mathcal{G}_n$ and $\mathcal{G}_n^{-1}$ are the only permutations requiring $n - 1$ reversals.

$\frac{1}{2}n + \frac{1}{2}$ is a lower bound on the diameter of our problem, and as we have indicated the algorithm of Watterson et al. [24] uses no more than $n-1$ reversals; yet we show there are permutations for which this algorithm performs arbitrarily poorly in ratio. Section 2 presents an approximation algorithm that does achieve a factor of 2 for unrestricted reversals.

Finally from the perspective of group theory, Even and Goldreich [6] and Jerrum [11] are interesting. Even and Goldreich show that, given a set of generators of a permutation group G and a permutation π, determining the shortest product of generators that equals π is NP-hard.[8] Their reduction implies that the problem remains NP-hard even if every generator is its own inverse, as is the case in our problem. Jerrum established that the problem is PSPACE-complete, and remains so when restricted to two generators.[9]

In our problem the generator set is fixed. Thus, while these complexity results give a feel for the problem, they do not imply the intractability of sorting by reversals. Nevertheless, we believe sorting by reversals is NP-complete. Section 5 indicates one possible direction for a proof.

Overview

In the next section we present an approximation algorithm for sorting by reversals. We show it never exceeds the minimum number by more than a factor of 2 and has a simple quadratic time implementation.

Section 3 develops an exact algorithm using the branch and bound technique. The lower bound uses a relaxation to maximum weight matchings and linear programming.

Section 4 presents results from experiments with an implementation of these algorithms. We study their performance on random permutations and permutations generated by a fixed number of random reversals.

Section 5 concludes with some open problems and conjectures.

2 An approximation algorithm

Perhaps the most natural algorithm for sorting by reversals, suggested by Watterson, Ewens, Hall and Morgan [24], is to bring element 1 into place, then element 2, and so on up to element n. Formally, at step i perform reversal $[i, \pi_i^{-1}]$ if $\pi_i \neq i$,. Once step $n-1$ is completed element n must be in position n, so this sorts any n-element permutation in at most $n-1$ reversals.

While it is likely there exist permutations for every n that require $n-1$ reversals [10], which if true means this algorithm is worst-case optimal, for specific instances the algorithm can perform arbitrarily poorly. Consider for example the permutation $(n\ 1\ 2\ \cdots\ n{-}1)$. Bringing 1 into place, then 2, and so on, uses $n{-}1$ reversals,

[8] Even and Goldreich also show that computing the diameter of G is NP-hard. Determining whether there exists a product equal to π is solvable in polynomial time [7].

[9] This result is best possible, since the case of a single generator can be solved in polynomial time. Jerrum also shows that the problem is polynomial time solvable for any of the standard sets of generators for the symmetric and alternating groups: all transpositions or 2-cycles, all adjacent transpositions, all transpositions adjacent on the circle, all 3-cycles, or all 3-cycles with all pairs of disjoint transpositions.

yet the permutation can be sorted in two steps: reverse $[1, n]$, then $[1, n-1]$. Thus this natural algorithm can produce a solution $\frac{1}{2}(n-1)$ times longer than the shortest, for arbitrarily large n. Using the idea of a breakpoint, also introduced in Watterson et al. [24], we show there is a simple algorithm guaranteed to use no more than twice the minimum number of reversals. To the best of our knowledge, this is the first constant-factor approximation for sorting by reversals.

In order to describe the algorithm, we first define some terminology.

A *breakpoint* of a permutation π is a pair of adjacent positions $(i, i+1)$ such that $|\pi_{i+1} - \pi_i| \neq 1$. In other words, $(i, i+1)$ forms a breakpoint if values π_i and π_{i+1} are consecutively increasing or decreasing. To handle the boundaries we let π_0 have the value 0, π_{n+1} have the value $n+1$, and allow i to range from 0 to n in the definition. Thus $(0, 1)$ is a breakpoint if $\pi_1 \neq 1$, and $(n, n+1)$ is a breakpoint if $\pi_n \neq n$. Notice that the identity permutation has no breakpoints, any other permutation has some breakpoint, and the number of breakpoints is at most $n+1$.

When $|\pi_{i+1} - \pi_i| = 1$, we say values π_{i+1} and π_i are *adjacent*, and write $\pi_{i+1} \sim \pi_i$.

A *strip* of π is an interval $[i, j]$ such that $(i-1, i)$ and $(j, j+1)$ are breakpoints, and no breakpoint lies between them. In other words, a strip is a maximal run of increasing or decreasing elements.

A reversal ρ affects the breakpoints of π only at the endpoints of ρ. (In the interior, ρ only makes an increasing pair (π_i, π_{i+1}) decreasing, and vice versa.) Let us write $\Phi(\pi)$ for the number of breakpoints in π, and for a given reversal ρ, let

$$\Delta\Phi(\pi) \;=\; \Phi(\pi) - \Phi(\pi \circ \rho).$$

Since a reversal $[i, j]$ changes the adjacency of only two points, namely $(i-1, i)$ and $(j, j+1)$, the only values $\Delta\Phi(\pi)$ can have are between -2 and 2. Since a solution must decrease the number of breakpoints from $\Phi(\pi)$ to zero, a *greedy strategy* is to choose a reversal that achieves the greatest decrease, or maximizes $\Delta\Phi(\pi)$. As any $\pi \neq \imath$ has a reversal with $\Delta\Phi(\pi) \geq 0$, we can always achieve a decrease of 2, 1, or 0.

Figure 2 specifies our greedy algorithm. The algorithm removes zero breakpoints when there are no reversals that remove one or two, so it is not obvious that it terminates. With the rule "favor reversals that leave decreasing strips" we are able to show that not only does the algorithm terminate, it exceeds the minimum by at most a factor of 2.

2.1 Quality of the approximation

In the following, a strip $[i, j]$ is *decreasing* iff $\pi_i, \pi_{i+1}, \ldots, \pi_j$ is decreasing. We consider a strip of one element to be decreasing, except for π_0 and π_{n+1}, which are always increasing. Thus the identity permutation forms one increasing strip extending from 0 to $n+1$.

Lemma 1 *Every permutation with a decreasing strip has a reversal that removes a breakpoint.*

Proof Consider the decreasing strip of π whose last element, π_i, is smallest. Element $\pi_i - 1$ must be in an increasing strip (else π_i is not smallest) that lies either to the left or to the right of the strip containing π_i, as illustrated in Figure 3. In either case, the indicated reversal removes at least one breakpoint. $\square$

```
algorithm Greedy(π)
    i := 0
    while π contains a breakpoint do
        i := i + 1
        Let ρ_i be a reversal that removes the most breakpoints of π,
            resolving ties among those that remove one breakpoint
            in favor of reversals that leave a decreasing strip.
        π := π ∘ ρ_i
    end
    return i, ρ_1ρ_2 ··· ρ_i
end
```

Figure 2 The greedy algorithm.

Lemma 2 *Let π be a permutation with a decreasing strip. If every reversal that removes a breakpoint of π leaves a permutation with no decreasing strips, π has a reversal that removes two breakpoints.*

Proof Again consider the decreasing strip of π containing the smallest element π_i. Case (b) of Figure 3 cannot occur, since ρ is a reversal that removes a breakpoint and leaves a decreasing strip. Thus the increasing strip containing $\pi_i - 1$ must be to the left of the strip containing π_i, as in case (a). Call the reversal of case (a), ρ_i.

Consider the decreasing strip of π whose first element, π_j, is largest. Element $\pi_j + 1$ must be in an increasing strip (else π_j is not largest) that is to the right of the strip containing π_j. Otherwise, a reversal analogous to case (b) removes a breakpoint and leaves a decreasing strip. Call the reversal for π_j that is analogous to ρ_i, ρ_j.

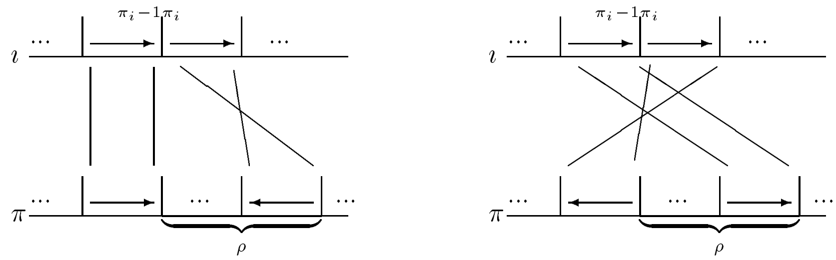
Notice that π_j must lie in interval ρ_i and $\pi_j + 1$ must lie outside, else ρ_i leaves a decreasing strip. Similarly π_i must lie in ρ_j and $\pi_i - 1$ outside, else ρ_j leaves a decreasing strip. The situation is as shown in Figure 4. Intervals ρ_i and ρ_j overlap.

We now argue that not only do ρ_i and ρ_j overlap, they must be the same interval. For suppose $\rho_i - \rho_j$ is not empty. If it contains a decreasing strip, reversal ρ_j leaves a decreasing strip, and if it contains an increasing strip, reversal ρ_i leaves a decreasing strip. Similarly interval $\rho_j - \rho_i$ must be empty, which implies interval $\rho_i = \rho_j$.

Since reversal ρ_i removes the breakpoint on its left and reversal ρ_j removes the breakpoint on its right, and as these breakpoints are distinct, reversal $\rho = \rho_i = \rho_j$ removes two breakpoints. □

Lemma 3 *The greedy algorithm sorts a permutation π with a decreasing strip in at most $\Phi(\pi) - 1$ reversals.*

Proof The proof is by induction on $\Phi(\pi)$. If π has a decreasing strip, $\Phi(\pi) \geq 2$. When $\Phi(\pi) = 2$, π has a unique reversal ρ that removes the two breakpoints and sorts π. Since Greedy will choose ρ it sorts π in one reversal, and the basis holds.



(a) (b)

Figure 3 A permutation π with a decreasing strip has a reversal ρ that removes a breakpoint. Element π_i is the smallest that lies in a decreasing strip.

Suppose the lemma holds for all π' with less than $\Phi(\pi)$ breakpoints. Since π has a decreasing strip, by Lemma 1 there is a reversal ρ that removes at least one breakpoint of π. Thus the first step of Greedy will transform π into a permutation π' with at most $\Phi(\pi) - 1$ breakpoints. If π' has a decreasing strip Greedy sorts it in at most $\Phi(\pi) - 2$ reversals by the induction hypothesis, which sorts π in at most $\Phi(\pi) - 1$ reversals.

Now consider a π' with no decreasing strips. We argue that $\Phi(\pi') = \Phi(\pi) - 2$. For suppose $\Phi(\pi') = \Phi(\pi) - 1$, the only other possibility. Since Greedy chooses a reversal that removes the most breakpoints, there must be no reversal that removes two. In other words, every reversal that removes a breakpoint removes exactly one breakpoint. Since in such an event Greedy chooses a reversal that leaves a decreasing strip whenever possible, every available reversal that removes a breakpoint must leave no decreasing strips. But by Lemma 2, this implies π has a reversal that remove two breakpoints, a contradiction. Thus $\Phi(\pi') = \Phi(\pi) - 2$.

Every reversal on a permutation with no decreasing strips creates a decreasing strip, which implies Greedy will transform π' to a permutation π'' with a decreasing strip. Moreover $\Phi(\pi'') \leq \Phi(\pi') = \Phi(\pi) - 2$. By induction, Greedy sorts π'' in at most $\Phi(\pi) - 3$ reversals. Since Greedy transformed π to π'' in two steps, this sorts π in at most $\Phi(\pi) - 1$ reversals. □

Theorem 1 *The greedy algorithm sorts every permutation π in at most $\Phi(\pi)$ reversals.*

Proof If π has a decreasing strip, by Lemma 3, Greedy sorts π within $\Phi(\pi)$ reversals. If π has no decreasing strip, any reversal chosen by Greedy transforms π to a permutation π' with a decreasing strip such that $\Phi(\pi') \leq \Phi(\pi)$. By Lemma 3, Greedy sorts π' in at most $\Phi(\pi') - 1$ reversals, which sorts π in at most $\Phi(\pi)$ reversals. □

Since $\Phi(\pi) \leq n + 1$, Theorem 1 implies that the greedy algorithm terminates in $O(n)$ iterations. In the next section we consider how to implement an iteration. Here we simply note that an iteration runs in polynomial time. An algorithm for an optimization problem that runs in polynomial time and delivers a solution whose value

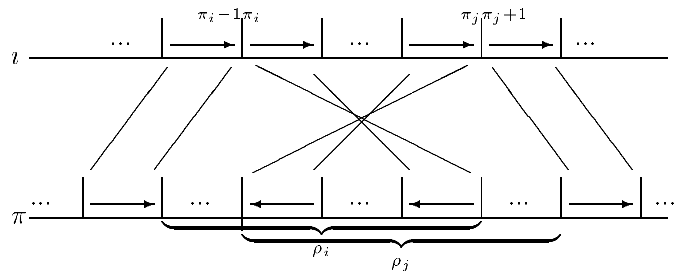


Figure 4 If every reversal that removes a breakpoint of π leaves a permutation with no decreasing strips, ρ_i and ρ_j must overlap. Elements π_i and π_j are the smallest and largest in decreasing strips.

is within a factor ϵ of optimal is known as an *ϵ-approximation algorithm*. An immediate consequence of Theorem 1 is the following.

Corollary 1 *The greedy algorithm is a 2-approximation algorithm for sorting by reversals.*

Proof Write $\mathrm{Opt}(\pi)$ for the minimum number of reversals to sort a permutation π, and $\mathrm{Greedy}(\pi)$ for the number taken by the greedy algorithm. Since a solution must remove all breakpoints and any reversal can remove at most two,

$$\mathrm{Opt}(\pi) \;\geq\; \frac{1}{2}\,\Phi(\pi) \;\geq\; \frac{1}{2}\,\mathrm{Greedy}(\pi).$$

$\square$

We do not know whether the bound of Corollary 1 is tight.

2.2 Time and space

How much time does an iteration of the greedy algorithm take? As there are in general $\binom{n}{2}$ reversals to consider, a naive implementation could take $O(n^2)$ time per iteration, or $O(n^3)$ total time. By considering the form of reversals that remove breakpoints we can find the greedy reversal for an iteration in $O(n)$ time, which yields an $O(n^2)$ time algorithm.

A reversal that removes two breakpoints must have both endpoints at breakpoints and must create two new adjacencies. Let us denote the left endpoint by $(i, i+1)$ and the right endpoint by $(j, j+1)$. Then reversal $[i+1, j]$ removes two breakpoints iff $(i, i+1)$ and $(j, j+1)$ are breakpoints, $\pi_i \sim \pi_j$, and $\pi_{i+1} \sim \pi_{j+1}$, as illustrated in Figure 5.

We can search for a reversal of this form as follows. Scanning π, we identify each breakpoint $(i, i+1)$. If this is the left end of such a reversal, there must be a

$$\pi \quad \ldots\; x \,\Big|\, \underbrace{y \quad \ldots \quad x'}_{\rho} \,\Big|\, y' \;\ldots$$

Figure 5 The form of a reversal that removes two breakpoints. Bars denote breakpoints, and $x \sim x'$ and $y \sim y'$.

position $j > i$ such that $(j, j+1)$ is a breakpoint, $\pi_j \sim \pi_i$, and $\pi_{j+1} \sim \pi_{i+1}$. There are two possible values for π_j, namely $\pi_i - 1$ and $\pi_i + 1$. Given π^{-1}, we can find the positions where $\pi_i - 1$ and $\pi_i + 1$ occur in $O(1)$ time. If either position meets the criteria above, we have found a reversal that removes two breakpoints. As there are $O(n)$ candidates for the left endpoint and π^{-1} can be computed in $O(n)$ time, this finds a reversal that removes two breakpoints (if one exists) in $O(n)$ time.

If there is no reversal that removes two breakpoints, the greedy algorithm considers reversals that remove one breakpoint. A reversal that removes one breakpoint must have one endpoint at the breakpoint it removes, while the other end may or may not lie at a breakpoint, as shown in Figure 6. Given π^{-1} we can find a reversal of form (a) or (b) in $O(1)$ time per breakpoint as above. The only question is how to efficiently determine whether the reversal leaves a decreasing strip.[10]

$$\pi \quad \ldots\; x \,\Big|\, \underbrace{y \quad \ldots \quad x'}_{\rho} \,\Big|\, z \;\ldots \qquad\qquad \pi \quad \ldots\; x \,\Big|\, \underbrace{x''' \quad \ldots \quad}_{\rho} \,\Big|\, x' x'' \,\Big|\; \ldots$$

(a) (b)

Figure 6 The form of a reversal that removes one breakpoint. In the figure, $x \sim x' \sim x'' \sim x'''$, and $y \not\sim z$. Mirror images of (a) and (b) are considered to be the same form.

Consider a reversal of form (a). Reversal $[i, j]$ leaves a decreasing strip iff

- $[1, i)$ contains a decreasing strip other than x, or
- $[i, j]$ contains an increasing strip[11] other than x', or
- $(j, n]$ contains a decreasing strip, or
- xx' is decreasing.

[10] Recall that the greedy algorithm breaks ties among reversals that remove one breakpoint by favoring reversals that leave decreasing strips.

[11] A strip is *increasing* if its elements are strictly increasing or it contains a single element. Thus a single element strip is both increasing and decreasing.

The only difficulty is in determining whether an interval contains an increasing strip or a decreasing strip in $O(1)$ time. We can solve this by forming an array, *down*$[i]$, that gives the position of the left end of the leftmost decreasing strip beginning at or to the right of position i. Then interval $[a, b]$ contains a decreasing strip iff *down*$[a] \leq b$. Similarly, we can form an array *up*$[i]$ that gives the left end of the leftmost increasing strip at or to the right of i. Both arrays can be computed from π in $O(n)$ time, for example by the recurrence

$$\mathrm{down}(i) \;=\; \begin{cases} n+1, & i > n; \\ i, & i \leq n,\ \pi_i \not\sim \pi_{i-1},\ \pi_i \not\sim \pi_{i+1}; \\ i, & i \leq n,\ \pi_i \not\sim \pi_{i-1},\ \pi_i \sim \pi_{i+1},\ \pi_i > \pi_{i+1}; \\ \mathrm{down}(i+1), & \text{otherwise.} \end{cases}$$

Thus if there exists a reversal of form (a) that removes one breakpoint, we can find it in $O(n)$ time whether we require that it leaves a decreasing strip or not. We can search for a reversal of form (b) in $O(n)$ time by the same technique.

If there is no reversal that removes a breakpoint, the greedy algorithm chooses a reversal that does not increase the number of breakpoints. One such reversal is $[i, \pi_i^{-1}]$, where i is the least position such that $\pi_i \neq i$. Notice this reversal cannot increase the number of breakpoints, since it always removes breakpoint $(i-1, i)$.

To summarize, we find a greedy reversal as follows.

(1) Compute π^{-1}, *down*, and *up*.
(2) Search for a reversal that removes two breakpoints.
(3) If none exists, search for a reversal that removes one breakpoint and leaves a decreasing strip.
(4) If none exists, search for a reversal that just removes one breakpoint.
(5) If none exists, bring the least out of place element into position.

Each step can be performed in $O(n)$ time using $O(n)$ space, which gives an $O(n^2)$ time implementation of the greedy algorithm. We suspect that the algorithm has an $O(n \log n)$ time implementation, but as we will see in Section 4, the approximation algorithm is hardly the dominant step in practice.

3 An exact algorithm

In the last section we obtained an algorithm that comes close to the optimum by applying a greedy strategy: of all reversals, select one that removes the most breakpoints. To obtain an algorithm that reaches the optimum we use a *branch and bound strategy*: consider all reversals, and eliminate those that cannot lead to an optimal solution.

Figure 7 shows the form of our branch and bound algorithm. We maintain three global variables: d^*, a dynamic upper bound on the solution value; r^*, an array of reversals that sort the permutation in d^* steps; and r, the series of reversals currently under consideration. At the start we initialize d^* and r^* to values obtained from an upper bound algorithm. The algorithm we use is essentially Greedy with a fixed-depth look-ahead, and is described in Section 3.3.[12]

[12] We use $x_1, \ldots, x_n := e_1, \ldots, e_n$ as shorthand for the parallel assignments $x_i := e_i$. Function UpperBound, like Greedy, returns two values: an integer, followed by a list of reversals.

After obtaining an upper bound we explore a tree of subproblems depth-first. Each invocation of Search corresponds to a node of the tree and is labelled with π, a permutation to be sorted, and d, the number of edges from the root to the node. Array r is maintained as a stack by Search, and holds the reversals on the path from the root to the current node. We chose a depth-first strategy for traversing the tree as this uses a polynomial amount of space even when the tree is of exponential size since space, not time, is often the limiting resource.

```
global d*, r*[1..n], r[1..n]

algorithm BranchAndBound(π)
    d*, r* := UpperBound(π)
    Search(π, 0)
    return d*, r*
end

algorithm Search(π, d)
    if π is the identity permutation then
        d*, r* := d, r
    else
        for each reversal ρ in order of decreasing ΔΦ do
            d′, π′ := d + 1, π ∘ ρ
            if d′ + LowerBound(π′) < d* then
                r[d′] := ρ
                Search(π′, d′)
            end
        end
end
```

Figure 7 The branch and bound algorithm.

Examining all reversals yields a very large tree: with $\binom{n}{2}$ children per node and a height of $n-1$, there are $O(n^{2n}/2^n)$ nodes. In Section 5 we state several conjectures on the form of a solution which, if true, reduce the children per node from $O(n^2)$ to $O(\Phi^2(\pi))$. Lacking a proof of these conjectures, the means we have to reduce the size of the tree is by ordering children and computing lower bounds. The algorithm of Figure 7 orders children by decreasing $\Delta\Phi$ on the assumption that the optimal solution uses reversals of greatest $\Delta\Phi$. By trying such reversals first we hope to quickly lower our upper bound; the earlier we lower the upper bound, the more subtrees we will prune. We now explain how the lower bound is computed.

3.1 A lower bound from matchings

As stated in the proof of Corollary 1, a simple lower bound on $\mathrm{Opt}(\pi)$ is $\lceil \Phi(\pi)/2 \rceil$. While this is sufficient to prove an approximation factor of 2, it is extremely weak. It assumes every breakpoint of π can be eliminated by a reversal that removes two breakpoints, which can rarely be achieved. To obtain a better bound we ask, for a given permutation, how many breakpoints can possibly be eliminated by reversals that remove two breakpoints?

A pair of breakpoints $p = (i, i+1)$ and $q = (j, j+1)$ with values $(\pi_i, \pi_{i+1}) = (x, y)$ and $(\pi_j, \pi_{j+1}) = (x', y')$ can be eliminated in one reversal iff $x \sim x'$ and $y \sim y'$. This holds whether p is to the left or to the right of q. The only requirement is that x and y occur in the same order as x' and y'.

Notice that such a reversal also effects other pairs of breakpoints. A pair with values (x, y) and (y', x'), which cannot be eliminated immediately because adjacent values are not in the same order, can be eliminated in one step if preceded by a reversal that contains exactly one breakpoint of the pair. Such a reversal transforms the pair to the preceding case.

In general, determining when a collection of $2m$ breakpoints can be eliminated by a sequence of m reversals appears difficult. (In Section 5 we conjecture it is NP-complete.) To obtain a lower bound that can be efficiently computed, we ignore dynamic information about the order and interaction of reversals. The static information we retain is simply the adjacency of values between breakpoints, which can be represented by a graph.

Figure 8 shows the construction. Each breakpoint of π is mapped to a vertex of $G(\pi)$. We place an edge between breakpoints p and q iff either of the above cases apply. Effectively, if two breakpoints may be eliminated by one reversal, possibly after a sequence of other reversals that eliminate two breakpoints, they share an edge. Note that the order of the two values at a breakpoint is not important for the construction.

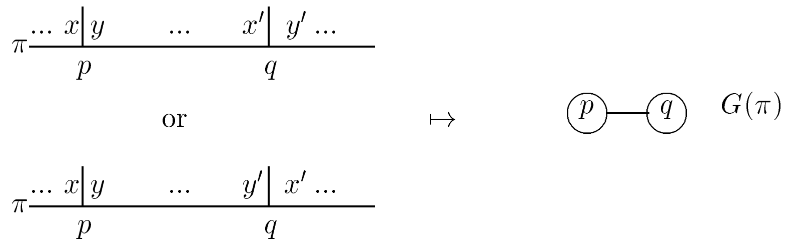


Figure 8 Breakpoints $\{x, y\}$ and $\{x', y'\}$ share an edge in $G(\pi)$ iff $x \sim x'$ and $y \sim y'$.

Since each edge models a reversal and performing the reversal removes both vertices, a series of reversals of π that each eliminate two breakpoints corresponds to a set of vertex-disjoint edges in $G(\pi)$. A set of vertex-disjoint edges is a *matching.* The key property of $G(\pi)$ is that the most reversals we can possibly perform on π that each remove a pair of breakpoints, without performing any intervening reversals that remove less than two breakpoints, is the size of a maximum cardinality matching of $G(\pi)$.

Let m be the number of vertices in a maximum cardinality matching of $G(\pi)$, or in other words twice the number of edges in the matching. How many reversals must be performed to remove the remaining $\Phi(\pi) - m$ breakpoints of π? The best we can do is to expend a reversal that removes one breakpoint to set up a reversal that removes two breakpoints. Notice that we cannot remove one breakpoint, then two, then two again, since a reversal that removes one breakpoint can affect only one additional breakpoint, so the third reversal must have been available from the start. This is impossible since the matching is of maximum cardinality. In short the best we can do is to remove three breakpoints in two reversals. This gives a lower bound of

$$\left\lceil \tfrac{1}{2}m \; + \; \tfrac{2}{3}\Big(\Phi(\pi) - m\Big) \right\rceil, \tag{1}$$

which has the extreme value $\lceil \frac{2}{3}\Phi(\pi) \rceil$.

We can construct $G(\pi)$ from π in $O(n)$ time. Certainly the $O(n)$ breakpoints of π can be determined in $O(n)$ time. Every breakpoint has a constant number of edges, since the only values adjacent to x are $x+1$ and $x-1$, and with the help of π^{-1} we can determine all edges in $O(n)$ time as well. Finally, a maximum cardinality matching of a graph with V vertices and E edges can be computed in $O(E\sqrt{V})$ time [15]. As we have said, V and E are $O(n)$. Thus we can evaluate the lower bound in $O(n^{3/2})$ time.

3.2 A family of lower bounds

One can improve the bound further by considering 3-tuples of breakpoints, 4-tuples of breakpoints, and so on.

Let us call a reversal that eliminates k breakpoints a k*-move.* Thus a 2-move is a reversal that eliminates 2 breakpoints, and a (-2)-move is a reversal that creates 2 breakpoints.

In general for $k \geq 3$ we define a k-move as follows. Over all permutations consider all series of reversals that eliminate k breakpoints. A k-move for $k \geq 3$ is a shortest series that eliminates k breakpoints given that no 2-, 3-, up to (k–1)-moves are available.

The following lemma characterizes the structure of a k-move.

Lemma 4 *For $k \geq 3$, a k-move is a series of $k-1$ reversals that decomposes into either*

(i) *a* 1*-move followed by a $(k-1)$-move, or*
(ii) *a* 0*-move followed by an i-move and a j-move where $i + j = k$.*

Proof Any series of reversals begins with a 2-, 1-, 0-, (-1)-, or (-2)-move. By definition a k-move for $k \geq 3$ cannot begin with a 2-move. Furthermore, any series that creates breakpoints is not among the shortest. Thus 1-moves and 0-moves are the only candidates for the first reversal in a k-move.

Consider a series that begins with a 1-move. The 1-move can change the values of at most two breakpoints, namely those at its endpoints. One of the two breakpoints is eliminated by the 1-move. The other can at best be eliminated in a $(k-1)$-move. Note that the $k-1$ breakpoints remaining cannot be eliminated by two or more higher-order moves as any second higher-order move would be initially available, contradicting the definition of a k-move.

Now consider a series that begins with a 0-move. This move can again affect the values of at most two breakpoints, thereby setting up at most two higher-order moves. The k-breakpoints again cannot be eliminated by three or more higher-order moves after performing the initial 0-move, since a third higher-order move would be initially available. That a k-move uses $k-1$ reversals follows by induction. □

With this decomposition we can characterize the breakpoints in a k-move.

Lemma 5 *For $k \geq 2$, the sets of values at the breakpoints eliminated by a k-move have the form*

$$\{x_1, x_2'\}\ \{x_2, x_3'\}\ \cdots\ \{x_{k-1}, x_k'\}\ \{x_k, x_1'\} \tag{2}$$

where $x_i \sim x_i'$ for $1 \leq i \leq k$.

Proof Notice that the lemma holds for $k = 2$, which corresponds to the picture of Figure 8. Assume then the lemma for all $k' \leq k$. We show it holds for $k + 1$.

By Lemma 4, a $(k+1)$-move decomposes into a 1-move followed by a k-move, or a 0-move followed by an i-move and a j-move. Consider the case of an initial 1-move.

This move eliminates one of the $k+1$ breakpoints and brings the remaining k breakpoints into the configuration of a k-move. By induction, the values at these k breakpoints have the form of (2). Notice that this form is unchanged by a rotation of the breakpoints, i.e. a renaming of the form $x_i \mapsto x_{i\oplus d}$ and $x_i' \mapsto x_{i\oplus d}'$ for any d, where $i \oplus d$ denotes $(i + d) \bmod (k) + 1$. Thus we may assume without loss of generality that the 1-move affects the values of (2) by bringing x_1, x_2' together into a breakpoint, and some other pair of values y, y' together to create an adjacency $y \sim y'$. This $(k+1)$-move then has the form

$$\{x_1, y'\}\ \{y, x_2'\}\ \{x_2, x_3'\}\ \cdots\ \{x_k, x_1'\},$$

which is the same form as (2).

Now consider the case of a 0-move that sets up an i-move and a j-move. By induction the i-move has the form

$$\{x_1, x_2'\}\ \{x_2, x_3'\}\ \cdots\ \{x_i, x_1'\},$$

and the j-move has the form

$$\{y_1, y_2'\}\ \{y_2, y_3'\}\ \cdots\ \{y_j, y_1'\}.$$

Without loss of generality assume the 0-move brings x_1, x_2' together and y_1, y_2' together. Since the form of (2) is unchanged by renaming $x_i \mapsto x_i'$ and $x_i' \mapsto x_i$, we may further assume that the 0-move brings these values together by touching breakpoints

$$\{x_1, y_1\}\ \{x_2', y_2'\}.$$

The $(k+1)$-move then has the form

$$\{x_2, x_3'\}\ \{x_3, x_4'\}\ \cdots\ \{x_i, x_1'\}\ \{x_1, y_1\}\ \{y_j, y_1'\}\ \{y_{j-1}, y_j'\}\ \cdots\ \{y_2, y_3'\}\ \{x_2', y_2'\}.$$

As the reader may verify, this is the same form as (2). □

We now describe how to construct a graph $H(\pi)$ from permutation π that allows us to efficiently identify sets of breakpoints of form (2) for small k. In the construction, breakpoints of π are mapped to vertices of $H(\pi)$, and pairs of breakpoints that share an adjacency, such as $\{x, a\}$ $\{x', b\}$ where $x \sim x'$, are mapped to edges as shown in Figure 9.

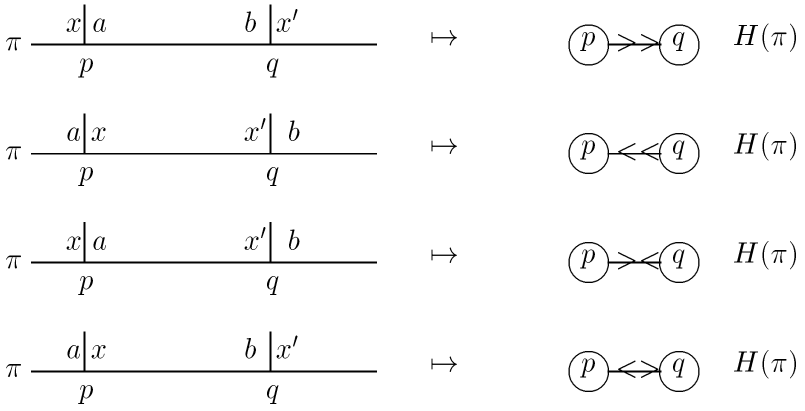


Figure 9 Construction of graph $H(\pi)$. Values x and x' are adjacent, and induce an edge that contributes to the in- or out-degree of p and the in- or out-degree of q, depending on whether the value is on the left- or right-side of the breakpoint.

Edges of $H(\pi)$ are directed, but not in the standard sense. An edge touching v and w contributes to either the in- or out-degree of v and w. In a directed graph, an edge (v, w) that contributes to the out-degree of v necessarily contributes to the in-degree of w, and vice versa. However in what we call a *bidirected graph* there are two more possibilities: (v, w) may contribute to the in-degree of both v and w, or to the out-degree of both v and w.

This gives rise to the four types of edges of Figure 9. We indicate the direction of an edge with double-ended arrows. When drawing an edge incident to v, we place an arrowhead pointing into v at the end touching v if the edge contributes to v's in-degree. Otherwise we direct the arrowhead out of v.

The utility of this construction is in the correspondence between cycles in the graph and k-moves on the permutation, as summarized in the following lemma. A *k-cycle* in a bidirected graph is a series of edges

$$(v_1, v_2)\ (v_2, v_3)\ \cdots\ (v_k, v_1)$$

such that the v_i are distinct, and every v_i has in- and out-degree 1.

Lemma 6 *The sets of breakpoints of π whose values have the form of k-moves are in one-to-one correspondence with the k-cycles of $H(\pi)$.*

Proof By Lemma 5, the values in a k-move have the form

$$\{x_1, x_2'\}\ \{x_2, x_3'\}\ \cdots\ \{x_k, x_1'\}.$$

In the i-th breakpoint $\{x_i, x'_{i\oplus 1}\}$, value $x'_{i\oplus 1}$ is adjacent to value $x_{i\oplus 1}$ of the $(i \oplus 1)$-th breakpoint. Whether x'_i is on the left or right of $x'_{i\oplus 1}$ in the i-th breakpoint, breakpoint i is linked in $H(\pi)$ to breakpoints $(i \oplus 1)$ and $(i \ominus 1)$ by edges that contribute exactly once to its in- and out-degree.

Similarly any k-cycle of $H(\pi)$ describes a set of k breakpoints with the property that every breakpoint of the set has values that are adjacent to values in both the preceding and succeeding breakpoints on the cycle. As every vertex of the cycle has in- and out-degree 1, these values by the construction are distinct. By Lemma 5, this is the form of a k-move. □

We now have the tools to generalize the lower bound of Section 3.1. In outline, we construct a hypergraph $G^{(k)}(\pi)$ whose vertices correspond to breakpoints, but whose edges are sets of up to k vertices that correspond to k'-moves for $k' \leq k$. A series of moves on π maps to a matching of $G^{(k)}$, where a matching of a hypergraph is a collection of vertex-disjoint edges. Choosing a k-move corresponds to performing a series of $k-1$ reversals. We weight edges by the number of reversals they represent and seek as before a maximum weight matching. However computing a maximum weight matching is in general NP-complete for hypergraphs [8]. We express the matching problem as an integer programming problem and relax the integrality constraint to obtain a linear programming problem. This gives a somewhat weaker lower bound that is computable in polynomial time.

This approach is summarized in the following theorem. The *k-girth* of a graph is the length of a shortest cycle of more than k edges. If the graph does not contain such a cycle, we define its k-girth to be $n+1$, where n is the number of vertices.

Theorem 2 *Let (V, E) be the graph $G^{(k)}(\pi)$, g be the k-girth of $H(\pi)$, and $\mathcal{L}_k(\pi)$ be the solution value of the linear program*

$$\begin{array}{ll} \text{minimize} & \frac{g-1}{g}\Phi(\pi) \;-\; \sum\limits_{e \in E} \frac{g-|e|}{g} x_e \\ \text{subject to} & \sum\limits_{e \mid v \in e} x_e \leq 1 \;\; \text{for all } v \in V, \\ & x_e \geq 0 \;\; \text{for all } e \in E. \end{array}$$

Then $\mathrm{Opt}(\pi) \geq \lceil \mathcal{L}_k(\pi) \rceil$.

Proof As in the lower bound of Section 3.1, the only characteristic of reversals that we consider are the values at their endpoints. This means we ignore the effects of reversals on their interior, namely that they change the relative order of elements.

This being the case, we first argue that to demonstrate our lower bound, the only series that we have to consider are those that do not create breakpoints. For suppose a reversal in a series creates a breakpoint. Eventually this breakpoint must be removed, and the best we can possibly do is to eliminate it with a 2-move, which allows us to remove one more breakpoint. Thus the best we can have achieved is to remove one breakpoint in two reversals. This is worse than any of the higher-order moves we consider in the lower bound, which always remove k breakpoints in $k-1$ reversals. Admittedly, the (-1)-move and subsequent 2-move by changing the relative order of elements may have made some advantageous moves possible in an actual series, but we have already accounted for such effects by ignoring the order of values at breakpoints in our graph representation.

Thus it suffices to consider series that do not create breakpoints. Such a series operates only on breakpoints in the original permutation by moving values from one breakpoint to another so as to create adjacencies. Decomposing the series into higher-order moves, every k'-move where $k' \le k$ maps to an edge of $G^{(k)}$, no matter where it occurs in the series. Moreover, the edges identified in $G^{(k)}$ are vertex disjoint, and form a matching M. Thus the number of reversals taken by k'-moves in the series where $k' \le k$ is

$$\sum_{e \in M} \Big(|e| - 1 \Big).$$

The remaining moves of the series are k'-moves where $k' > k$. By Lemma 6, every one of these moves maps to a cycle of $H(\pi)$, so the smallest $k' > k$ for which the series contains a k'-move is at least g, the k-girth of $H(\pi)$. Thus the number of reversals taken by k'-moves in the series where $k' > k$ is at least

$$(g-1)\left\lfloor \Big(\Phi(\pi) - \sum_{e\in M} |e|\Big) \Big/ g \right\rfloor \;+\; \Big(\Phi(\pi) - \sum_{e\in M} |e|\Big) \bmod g,$$

which is

$$\left\lceil \tfrac{g-1}{g}\Big(\Phi(\pi) - \sum_{e\in M} |e|\Big) \right\rceil.$$

Thus the total number of reversals in an optimal series is at least

$$\min_M \left\lceil \sum_{e\in M} \Big(|e| - 1\Big) \;+\; \tfrac{g-1}{g}\Big(\Phi(\pi) - \sum_{e\in M} |e|\Big) \right\rceil,$$

which is

$$\left\lceil \min_M \left\{ \tfrac{g-1}{g}\Phi(\pi) \;-\; \sum_{e\in M} \tfrac{g-|e|}{g} \right\} \right\rceil.$$

This is equivalent to finding a matching in $G^{(k)}$ of maximum total weight where the weight of an edge e is $\frac{g-|e|}{g}$. We can express this as an integer programming problem. For each edge e of $G^{(k)}$ we have a variable x_e that takes on the values 0 or 1. Selecting e is encoded by assigning x_e the value 1. We can ensure that the assignment represents a matching by requiring

$$\sum_{e \mid v \in e} x_e \;\le\; 1$$

for every vertex v. Extending the domain of x_e to real values between 0 and 1 results in the linear programming problem of the theorem. $\square$

Notice that the lower bound of Theorem 2 has the extreme value $\lceil \frac{g-1}{g}\Phi(\pi)\rceil$. This can be as large as $\Phi(\pi)$ even for small k, which meets the upper bound of Theorem 1.

How much time does it take to evaluate the lower bound $\mathcal{L}_k$? There are three tasks: (1) constructing H and computing its k-girth, (2) constructing $G^{(k)}$ and its associated linear program, and (3) solving the linear program.

Constructing H takes time $O(n)$. There are $O(n)$ breakpoints, and each breakpoint has at most four in-edges and at most four out-edges, which can be identified in $O(1)$ time per edge using π^{-1}.

We can compute the k-girth of H in $O(4^k n^2)$ time as follows. A shortest cycle of more than k edges that contains a fixed vertex v is a path P of k edges from v to some vertex w, followed by a shortest path from w to v that does not visit any other vertices on P. Paths in a bidirected graph such as H alternate in- and out-edges: if we enter a vertex by an in-edge, we must leave by an out-edge, and vice versa. As every vertex of H has in- and out-degree at most four, there are at most 4^k paths of length k from a fixed vertex v. For each path P, we can mark its vertices and compute a shortest path from its end w back to v, taking care not to include marked vertices. This shortest path can be found by a breadth-first search from w in $O(n)$ time. Repeating for all start vertices v, all paths P, and recording the minimum over all cycles found takes time $O(4^k n^2)$.

Similarly, we can construct the edges of $G^{(k)}$ in $O(4^k n)$ time by enumerating the cycles of H of k or fewer edges in a depth-first search.[13] Space for all edges is $O(k4^k n)$.

The resulting linear program has $O(4^k n)$ variables and $O(n)$ constraints. Writing $L(a, b)$ for the time to solve a linear program of a variables and b constraints, the linear programming problem takes $O(L(4^k n, n))$ time. This dominates the time to compute the lower bound. Thus for arbitrary but fixed k, $\mathcal{L}_k$ can be computed in $O(L(n, n))$ time.

3.3 A family of upper bounds

As well as a lower bound on the solution value our exact algorithm requires an upper bound. The simplest approach is to use the approximation algorithm of Section 2, but for large n this gives too weak a bound to prune away much of the search tree.

Consider however a series of k reversals that removes the most breakpoints among series of that length. The greedy strategy of the approximation algorithm is really based on the observation that once we are k reversals away from sorting a permutation, such a series is optimal. Greedy corresponds to the case $k = 1$.

One can find such a series by looking ahead k reversals, and this search can be made tractable by again employing branch and bound. The basic form of the computation is identical to BranchAndBound except that the recursion is stopped at depth k. The two requirements are a lower bound on the number of breakpoints that can be eliminated in a series of k reversals, and a method for computing an upper bound on the number eliminated by an optimal extension of a partial series.

We can compute the lower bound by running Greedy. Computing an upper bound is a little more difficult but can be tackled by the methods of the previous section, as summarized in the following theorem.

[13] Note that determining the edges of $G^{(k)}$ by examining all k-sets would take $O(n^k)$ time.

Theorem 3 *Let* (V, E) *be the graph* $G^{(k+1)}(\pi)$ *and* $\mathcal{U}_k(\pi)$ *be the solution value of the linear program*

$$\begin{array}{ll} \text{maximize} & k + \sum_{e \in E} x_e \\ \text{subject to} & \sum_{e \in E} \Big(|e| - 1\Big) x_e \;\leq\; k, \\ & \sum_{e | v \in e} x_e \leq 1 \;\; \text{for all } v \in V, \\ & x_e \geq 0 \;\; \text{for all } e \in E. \end{array}$$

Then $\lfloor \mathcal{U}_k(\pi) \rfloor$ *is an upper bound on the number of breakpoints of* π *that can be eliminated in* k *reversals.*

Proof The proof is similar to that of Theorem 2.

Given an allotment of k reversals, we want to eliminate as many breakpoints as possible. Suppose we consider all k'-moves for $k' \leq k+1$. Packing these k'-moves into the k reversals corresponds to finding a matching M of $G^{(k+1)}$ satisfying

$$\sum_{e \in M} \Big(|e| - 1\Big) \;\leq\; k,$$

since by Lemmas 4 and 6 each edge e of M represents $|e| - 1$ reversals. The number of breakpoints eliminated by these reversals is

$$\sum_{e \in M} |e|.$$

Having used $\sum_{e \in M}(|e|-1)$ reversals, we have $k - \sum_{e \in M}(|e|-1)$ remaining in our allotment. Notice that any unconsidered k'-move, which must have $k' \geq k+2$, will not completely fit in our allotment of k reversals since by Lemma 4 such a move takes at least $k+1$ reversals. Lemma 4 also implies that any prefix of such a k'-move that is packed into our allotment can on average remove at most one breakpoint per reversal. Thus the number of breakpoints eliminated by the reversals remaining in our allotment is at most

$$k \;-\; \sum_{e \in M} \Big(|e| - 1\Big).$$

Adding the number of breakpoints eliminated by the reversals in M to the number of breakpoints eliminated by the remaining reversals, the total number eliminated within k reversals is at most

$$k \;+\; \sum_{e \in M} 1.$$

Expressing this matching problem as an integer program and extending its domain to the reals results in the linear programming problem of the theorem. □

For fixed k the time and space to compute upper bound $\mathcal{U}_k$ is the same as for lower bound $\mathcal{L}_k$, which is $O(L(n,n))$ time and $O(n)$ space.

Given that we can find a series of k reversals that removes the most breakpoints, how should we piece together a solution from these series? One extreme is to perform only the first reversal of the series, arrive at a new permutation, and again look ahead k reversals. The other extreme is to execute all k of the series. We call the number of reversals that are performed from a series the *follow-through* of the algorithm.

One might predict that a follow-through of one reversal is best, since this retains the maximum flexibility. In our experience however this performed unexpectedly poorly in the sense that the final solution tended to degrade as the look ahead was increased beyond a critical value. While a thorough analysis of this phenomenon is beyond the scope of our paper, we offer the following explanation.

In an optimal solution to the global problem there are a fixed number of 2-moves. As the look ahead is increased, these 2-moves are discovered and bubbled towards the front of the solution until they appear within the series. In order to achieve these 2-moves, however, it is often necessary to perform slightly worse moves early on. (This is why a greedy strategy is not optimal.) Thus with a follow through of one reversal, we end up performing an initial move that is somewhat worse than if we had not looked so far ahead. Of course if we follow through with this series, we will be doing better in the end. But upon looking ahead from our new state we see there is an alternate series whose final move is even better, and we switch to the first move of this new series. By never following through on a series we end up making a sequence of poorer moves without ever reaping their benefit.

We also remark that looking ahead farther does not in general guarantee a better solution. After looking ahead $k+1$ reversals we simply end up in some state. Unless the permutation can be sorted in $k+1$ reversals, this state does not necessarily lead to a shorter solution than the state we arrive at after looking ahead only k reversals.

Nevertheless, choosing a follow-through equal to the look-ahead did have the desired property that the quality of the solution tended to improve as the look-ahead was increased, and this is the value that we chose for the experiments of the next section. A follow-through of k reversals also has the advantage of minimizing the total number of invocations of the branch and bound procedure. We simply note that it would be interesting to investigate other compromise values such as a follow-through of $\frac{k}{2}$.

To summarize, our exact algorithm constructs a solution in three stages, the first two of which are interleaved. The first stage runs the greedy algorithm to bound the number of breakpoints that can be eliminated within the look-ahead. The second stage improves this greedy solution by branch and bound to a fixed depth to obtain a series that is optimal within the look-ahead. Successive locally optimal series are concatenated to obtain a solution that upper bounds the global problem. The final stage improves this solution to a global optimum by a full branch and bound computation, now that a good upper bound is known.

This bootstrapping approach has proven to be quite effective, as is discussed in the next section.

4 Computational results

To examine the effectiveness of these ideas we tested a full implementation of the exact and approximation algorithms on biological and simulated data. The implementation comprises approximately 9,500 lines of C, of which roughly 2,500 lines are a sparse linear programming package.

An unusual aspect of the code is the manipulation of the bidirected graphs of Section 3.2. These graphs are sufficiently different from standard directed and undirected graphs to make the correct implementation of simple computations like shortest paths and cycle enumeration suprisingly tricky. Without going into detail, we note that a straightforward translation of the standard breadth-first search algorithm for single-source shortest paths is not correct, since a vertex can be reached in two different ways from the source: once by paths that end with an in-edge and once by paths that end in an out-edge. This means that essentially two distances are maintained for a vertex: an in-edge distance and an out-edge distance.

We also note that the code enumerates cycles of length at most k using the method outlined in Section 3.2, which can spend time exponential in k between reporting cycles. The algorithm of Johnson [12] for enumerating the cycles of a directed graph spends time linear in the size of the graph between reporting each cycle, and it would be interesting to see whether this algorithm can be adapted to our application.

In all we tested the implementation on four types of data: all permutations on a fixed number of elements, published gene order data from the biology literature, random permutations, and permutations generated by scrambling the identity with a fixed number of random reversals.

In the first set of experiments, which served as a good shake down of the program, we ran the implementation to optimality on all permutations of up to 8 elements. The exact distribution of reversal distance derived from these tests is given in Table 1. Note that the data supports Gollan's conjecture: for each $n > 2$ there exactly two permutations requiring $n - 1$ reversals, and these are $\mathcal{G}_n$ and its inverse.

Table 1 The number of permutations on n elements at distance d from the identity.

	n						
d	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1
1	1	3	6	10	15	21	28
2		2	15	51	127	263	483
3			2	56	390	1,562	4,635
4				2	185	2,543	16,445
5					2	648	16,615
6						2	2,111
7							2

On these experiments we also measured the worst case performance ratio of the approximation algorithm. On permutations of up to 8 elements the maximum ratio is $\frac{8}{5}$, which is achieved on permutation (4 7 2 6 8 5 3 1).

As an illustration of the algorithm on biological data we give the example of Figure 10. This permutation gives the order of the 36 genes that are common to the

12 $\boxed{\text{31 34 28 26 17 29 4 9 36 18 35 19 1 16 14 32 33 22 15 11 27 5 20 13}}$ 30 23 10 6 3 24 21 8 25 2 7

12 13 20 5 27 11 15 22 33 32 14 16 1 19 $\boxed{\text{35 18}}$ 36 9 4 29 17 26 28 34 31 30 23 10 6 3 24 21 8 25 2 7

12 13 20 $\boxed{\text{5 27 11 15 22 33 32 14 16 1 19 18 35 36 9}}$ 4 29 17 26 28 34 31 30 23 10 6 3 24 21 8 25 2 7

12 13 20 $\boxed{\text{9 36 35 18 19 1 16 14 32 33 22 15 11 27 5 4 29 17 26 28 34 31 30 23 10 6 3 24 21}}$ 8 25 2 7

12 13 20 21 24 $\boxed{\text{3 6 10 23 30 31 34 28 26 17 29 4 5 27 11 15 22 33 32 14 16 1 19 18 35 36 9 8 25}}$ 2 7

$\boxed{\text{12 13 20 21 24 25 8 9 36 35 18 19 1 16 14 32 33 22 15}}$ 11 27 5 4 29 17 26 28 34 31 30 23 10 6 3 2 7

15 $\boxed{\text{22 33 32 14 16 1 19 18 35 36 9 8 25 24}}$ 21 20 13 12 11 27 5 4 29 17 26 28 34 31 30 23 10 6 3 2 7

$\boxed{\text{15 24 25 8 9 36 35 18 19 1}}$ 16 14 32 33 22 21 20 13 12 11 27 5 4 29 17 26 28 34 31 30 23 10 6 3 2 7

1 19 18 $\boxed{\text{35 36 9 8 25 24 15 16 14 32 33 22 21 20 13 12 11 27 5 4 29 17 26 28}}$ 34 31 30 23 10 6 3 2 7

1 19 18 $\boxed{\text{28 26 17}}$ 29 4 5 27 11 12 13 20 21 22 33 32 14 16 15 24 25 8 9 36 35 34 31 30 23 10 6 3 2 7

1 $\boxed{\text{19 18 17 26 28 29 4 5 27 11 12 13}}$ 20 21 22 33 32 14 16 15 24 25 8 9 36 35 34 31 30 23 10 6 3 2 7

1 $\boxed{\text{13 12 11 27 5 4 29 28 26 17 18 19 20 21 22 33 32}}$ 14 16 15 24 25 8 9 36 35 34 31 30 23 10 6 3 2 7

1 $\boxed{\text{32 33 22 21 20 19 18 17 26 28 29 4 5 27 11 12 13 14 16 15 24 25 8 9 36 35 34 31 30 23 10 6 3 2}}$ 7

1 2 3 6 $\boxed{\text{10 23 30 31 34 35 36}}$ 9 8 25 24 15 16 14 13 12 11 27 5 4 29 28 26 17 18 19 20 21 22 33 32 7

1 2 3 6 $\boxed{\text{36 35 34 31 30 23 10 9 8 25 24 15 16 14 13 12 11 27 5 4 29 28 26 17 18 19 20 21 22 33 32 7}}$

1 2 3 $\boxed{\text{6 7 32 33 22 21 20 19 18 17 26 28 29 4}}$ 5 27 11 12 13 14 16 15 24 25 8 9 10 23 30 31 34 35 36

1 2 3 4 29 28 26 $\boxed{\text{17 18 19 20 21 22 33 32 7 6 5 27 11 12 13 14}}$ 16 15 24 25 8 9 10 23 30 31 34 35 36

1 2 3 4 29 28 26 14 13 12 11 $\boxed{\text{27 5 6 7 32 33 22 21 20 19 18 17 16 15 24 25 8 9 10}}$ 23 30 31 34 35 36

1 2 3 4 29 28 26 $\boxed{\text{14 13 12 11 10 9 8 25 24}}$ 15 16 17 18 19 20 21 22 33 32 7 6 5 27 23 30 31 34 35 36

1 2 3 4 29 28 26 $\boxed{\text{24 25 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 33 32 7 6 5 27}}$ 23 30 31 34 35 36

1 2 3 4 $\boxed{\text{29 28 26 27 5 6 7 32 33 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 25 24 23}}$ 30 31 34 35 36

1 2 3 4 $\boxed{\text{23 24 25 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 33 32 7 6 5}}$ 27 26 28 29 30 31 34 35 36

1 2 3 4 5 6 7 $\boxed{\text{32 33 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8}}$ 25 24 23 27 26 28 29 30 31 34 35 36

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 $\boxed{\text{33 32 25 24 23 27 26 28 29 30 31}}$ 34 35 36

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 $\boxed{\text{31 30 29 28 26 27 23 24 25}}$ 32 33 34 35 36

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 $\boxed{\text{25 24 23}}$ 27 26 28 29 30 31 32 33 34 35 36

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 $\boxed{\text{27 26}}$ 28 29 30 31 32 33 34 35 36

Figure 10 Near-optimal solution for gene orders from the mitchondrial genomes of mammals and the flatworm *Ascaris suum*. This solution of 27 reversals is within 2 reversals of optimal.

linearized mitochondrial genomes of mammals [3] and the flatworm *Ascaris suum* [25]. While we have been able to solve other permutations arising from mitochondria data in a small amount of time,[14] this 36 element permutation has proven extremely difficult to solve to optimality. The near-optimal solution of the figure was found after 24 seconds of computation on a 33 Mhz Silicon Graphics Iris 4D/300GTX. The lower and upper bounds from this run, of 25 and 27 reversals, were found using a lower bound family of 6 and an upper bound look ahead of 5 reversals.[15] The search tree during look-ahead had a maximum size of 2,234 nodes, where this counts all nodes at

[14] Computing the inversion distance between the mitochondrial genomes of mammals and the yeast *Schizosaccharomyces pombe* took 3.5 minutes (29 genes and 19 inversions), and between mammals and the fly *Drosophila yakuba* took 1.7 minutes (37 genes and 16 inversions). We make no claim for the biological significance of the particular solutions found, though the inversion distance tends to reflect evolutionary divergence.

[15] As described in Section 3.3, the follow-through for the upper bound was equal to the look-ahead in all experiments.

which a linear programming problem was solved (including nodes that were pruned by the lower bound). From the difference between the upper and lower bounds we know that this solution is within 2 reversals of optimal. A family of 10 and a look-ahead of 8, which required a search of 408,653 nodes and terminated after 7.5 hours of computation, failed to improve the bounds by even one reversal.

The limit of what we can reliably solve to optimality is around 30 elements. Table 2 gives the running time and search tree size to solve a sample of 10 random permutations on 30 elements to optimality. Though the lower bound family and upper bound look-ahead were the same across these runs, the execution time and search tree size varied by three orders of magnitude. Since the average running time and tree size are skewed by these outliers we also give median values, which are more representative of the sample.

Table 2 Running time and search tree size for exact solution of random permutations on 30 elements.

				Tree size							
Running time				Upper bound algorithm				Exact algorithm			
min	max	med	avg	min	max	med	avg	min	max	med	avg
0:05	43:48	1:29	6:53	228	372,563	11,419	45,393	0	8,291	29	843

Lower bound family is 8, upper bound look-ahead is 6 reversals, and sample size is 10 permutations. Running time is the total time for the exact algorithm in minutes, which includes the time to compute the upper bound. Tree size for the upper bound algorithm is the maximum size of the search tree during look-ahead. The average reversal distance of the sample is 20.6.

That we could solve these instances to optimality is due to the tightness of the bounds. Over the 10 permutations the maximum difference between the upper and lower bound was 2 reversals, and in each case the exact solution value was equal to the lower bound. Except for the one permutation on which the exact algorithm examined over 8,000 nodes, a common picture emerged from these runs. Optimal solution involves the upper bound algorithm exploring a rather large tree to find a solution that is within one or two reversals of optimal; with this solution in hand the exact algorithm hugs the left chain of its search tree to make up the difference to the lower bound, which is exact.

The effect of varying the family and look-ahead is shown in Table 3. On random permutations of 50 elements the average lower bound reached a maximum of roughly 36 reversals at family 6, and the average upper bound did not improved much on 38 reversals beyond look ahead 5.[16] Consequently these values of family 6 and look-ahead 5 were used in the remaining experiments.

[16]Notice that the average upper bound actually increased at look- ahead 8.

Table 3 Lower bound L, upper bound U, number of cycle sets C, and tree size T at various families and look-aheads for random permutations on 50 elements.

	L	U	C	T
	avg	avg	med	med
1	26.0	43.7	0	917
2	32.1	40.8	4	2,907
3	34.5	39.5	14	4,312
4	35.5	39.1	40	24,242
5	35.7	38.4	119	36,676
6	35.9	38.2	353	50,000
7	35.9	38.0	910	50,000
8	35.9	38.9	2,464	50,000

Maximum tree size is 50,000 nodes and sample size is 10 permutations. Row k gives the lower bound for family k, the upper bound for look-ahead k, the number of cycle sets of at most k breakpoints for the lower bound, and the maximum tree size for the upper bound during look-ahead.

The search tree for these runs was limited to 50,000 nodes. Once this limit was exceeded, the best series known within the look-ahead was used to form the upper bound. Column T gives the median tree size for the sample, where the tree size of a run is the maximum size of the search tree during look-ahead. For a look-ahead of 6 or more the majority of runs had a tree size meeting the limit. Column C gives the median number of cycle sets on k or few breakpoints, which is equivalent to the number of variables in the linear program for the lower bound at family k. The number of constraints on these variables is essentially the number of breakpoints in the permutation.

In the third set of experiments we studied the quality of the approximation algorithm and the upper bound algorithm on random permutations. Results are given in Table 4, where *dev* denotes the standard deviation. Note that the maximum difference between the upper and lower bound, which is what limits the range of optimal solution, was at most 2 reversals for n up to 30, while the average difference between the bounds was around 2.5 reversals for n up to 50. This suggests that while we can find optimal solutions for at most around 30 elements, we can find near-optimal solutions of acceptable quality for up to 50 elements.

The average performance ratio of the approximation algorithm for the sample was around $\frac{5}{4}$, while the worst ratio was less than $\frac{4}{3}$.

In the final set of experiments we were interested in the following question: How well does reversal distance recover the actual number of reversals applied to a permutation? To answer this we generated permutations by scrambling the identity with k random reversals, taking care not to reverse single elements. Table 5 gives upper and lower bounds on the reversal distance for permutations on 100 elements with up to 100 random reversals. For $k < \frac{1}{2}n$ the average upper bound estimated k to within one reversal, but for $k \geq \frac{1}{2}n$, it increasingly underestimated k.

Notice that when both endpoints of all reversals in a series of k reversals fall at new

Table 4 Lower bound L, upper bound U, approximation A, and tree size T for random permutations on n elements.

	L		U		$U-L$		A		$A\,/\,L$		T
n	avg	dev	avg	dev	avg	max	avg	dev	avg	max	med
10	6.0	1.1	6.0	1.1	0.0	0	6.1	1.0	1.02	1.20	0
20	12.6	1.2	13.2	1.5	0.6	2	14.8	1.5	1.18	1.31	517
30	20.8	0.9	21.8	1.6	1.0	2	24.9	1.5	1.20	1.25	4,615
40	28.5	1.1	30.3	1.6	1.8	4	33.8	2.0	1.19	1.24	23,712
50	35.9	1.4	38.4	1.8	2.5	5	43.7	2.7	1.22	1.31	36,676
60	43.6	1.1	46.7	1.5	3.1	5	52.6	1.6	1.21	1.26	50,000
70	51.7	1.3	56.7	2.4	5.0	8	63.4	3.0	1.23	1.30	50,000
80	58.9	1.0	64.5	2.0	5.6	8	72.3	2.3	1.23	1.26	50,000
90	67.6	1.4	74.1	2.1	6.5	8	83.2	1.8	1.23	1.25	50,000
100	74.2	1.1	82.4	2.6	8.2	10	91.9	2.7	1.24	1.27	50,000

Lower bound family is 6, upper bound look-ahead is 5 reversals, maximum tree size is 50,000 nodes, and sample size is 10 permutations. Approximation A is the number of reversals from the greedy algorithm. Tree size T is the maximum tree size for the upper bound during look-ahead.

Table 5 Upper bound U and lower bound L for permutations on 100 elements generated by k random reversals.

	U		L	
k	avg	dev	avg	dev
5	5.0	0.0	5.0	0.0
10	10.0	0.0	10.0	0.0
15	15.2	0.4	15.0	0.0
20	19.5	0.9	19.4	1.0
25	25.0	0.9	24.5	0.9
30	30.0	1.3	29.3	1.0
35	34.6	2.1	33.5	1.8
40	39.0	1.6	37.3	2.0
45	43.9	2.2	42.5	2.2
50	47.3	2.0	45.5	1.4
55	51.8	3.0	48.5	2.8
60	54.2	2.7	50.7	2.0
65	58.5	2.6	54.3	2.5
70	60.0	4.3	56.0	3.1
75	62.0	2.8	57.7	2.4
80	64.9	3.9	60.4	2.8
85	67.0	2.6	61.9	2.0
90	69.8	3.0	64.9	2.2
95	72.2	2.2	65.1	1.5
100	72.0	2.9	65.3	1.7

Upper bound look-ahead is 5 reversals, lower bound family is 6, maximum tree size is 50,000 nodes, and sample size is 10 permutations.

positions, the reversal distance for the resulting permutation is precisely k.[17] Since a permutation on n elements can have at most $n+1$ breakpoints, the maximum number of reversals for which this can happen is $\frac{1}{2}(n+1)$, which may partly explain why we observe a change in behavior around $k = \frac{1}{2}n$.

5 Conclusion

Algorithmic study of the reversal distance between permutations is in its earliest stages. We have presented two algorithms: a greedy approximation algorithm and a branch and bound exact algorithm. By analyzing the $\Delta\Phi$-sequence of the greedy algorithm we were able to show it achieves a worst-case approximation factor of 2, and by applying matchings, shortest paths, and linear programming we derived a class of nontrivial lower bounds for our exact algorithm.

Experiments with the exact algorithm indicate that we can solve random permutations up to around 30 elements to optimality usually in a few minutes, and that for permutations generated by k random reversals, the average solution value is a good estimate of the number of reversals for k less than half the number of elements.

We close with some conjectures and lines for future research.

Further research

One question we would like to resolve is the computational complexity of sorting by reversals. We conjecture it is NP-complete, and believe the crux of the problem is the following.

Conjecture 1 *Deciding* $\mathrm{Opt}(\pi) \leq \frac{1}{2}\Phi(\pi)$ *is NP-complete.* □

Notice that this special case has additional structure—for instance every breakpoint must be paired with another in $G(\pi)$—which should simplify a proof or disproof.

To design a more efficient exact algorithm we need theorems on the structure of a solution. In particular do we really need to consider all $\binom{n}{2}$ reversals to find an optimal solution?

It is natural to think we can throw out reversals that cut strips, since such a reversal separates elements that will have to be joined together later. Unfortunately this is incorrect. Permutation (3 4 1 2) for example requires three reversals if we do not cut strips, yet it can be sorted in two reversals as follows.

$$\big|3\ 4\big|1\ 2\big| \;\vdash\; 1\big|4\ 3\ 2\big| \;\vdash\; 1\ 2\ 3\ 4$$

Nevertheless, we believe the following is true.

Conjecture 2 *Every permutation has an optimal solution that does not cut strips other than at their first or last element.* □

---

[17] This follows because k is a lower bound as well as an upper bound on the reversal distance: the number of breakpoints divided by two, which lower bounds reversal distance, is in this situation equal to k.

If true, this reduces the number of candidate reversals from $O(n^2)$ to $O(\Phi^2(\pi))$. Our belief in Conjecture 2 is based on the following.

Conjecture 3 *Every permutation has an optimal solution that never increases the number of breakpoints.* □

Another tempting idea is to decompose the permutation into knots, where a *knot* of π is an interval $[a, b]$ such that

- $\pi(i) \in [1, a)$ for $i \in [1, a)$,
- $\pi(i) \in [a, b]$ for $i \in [a, b]$, and
- $\pi(i) \in (b, n]$ for $i \in (b, n]$,

and then solve knots independently. This is appealing since a permutation can always be sorted without an element ever crossing a knot. Perhaps suprisingly, it is also not optimal. Permutation (2 1 5 6 3 4) for example has knots $[1, 2]$ and $[3, 6]$, and takes four reversals if we solve the knots separately. Yet by working on them together, the permutation can be sorted in three reversals, as shown below.

$$\big|2\ 1\big|5\ 6\big|3\ 4\big| \ \vdash\ \big|6\ 5\big|1\ 2\ 3\ 4\big| \ \vdash\ \big|4\ 3\ 2\ 1\big|5\ 6 \ \vdash\ 1\ 2\ 3\ 4\ 5\ 6$$

Finally, we note that the suboptimality of the greedy algorithm is not because it arbitrarily chooses among moves that appear equally good. It is in fact necessary to consider 1-moves when 2-moves are available. Permutation (5 6 2 1 8 7 3 4) for instance has one 2-move, $[3, 6]$, and taking this move leads to a solution of four reversals. Yet the permutation can be sorted in three reversals as follows.

$$\big|5\ 6\big|2\ 1\big|8\ 7\big|3\ 4\big| \ \vdash\ 1\ 2\big|6\ 5\big|8\ 7\big|3\ 4\big| \ \vdash\ 1\ 2\big|6\ 5\ 4\ 3\big|7\ 8 \ \vdash\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8$$

Before concluding, we remark that the biological motivation for reversal distance suggests several variations. Here we identify five aspects of the problem that may be varied.

(1) The *linear/circular variation.* Some organisms and organelles have circular chromosomes, for which the data is the order of genes around a circle. In that case cyclic shifts of a permutation are isomorphic, which means $(\pi_1 \cdots \pi_n)$ is equivalent to $(\pi_i \cdots \pi_n\ \pi_1 \cdots \pi_{i-1})$. A solution is a series of reversals and cyclic shifts that transform one permutation to another, and we seek a solution with the fewest reversals.

(2) The *signed/unsigned variation.* A gene is not a point on a line or circle; it is a region of sequence. The sequence has a reading direction, and a reversal reverses not only the order of genes but their direction as well.

When two chromosomes are compared, information may be availabel on gene direction as well as order. We can encode this information by associating a sign with each element: positive for the forward direction and negative for the reverse. A reversal then changes the sign of elements it reverses.

In the signed case we have two signed permutations, and we seek a shortest series of reversals that transform one into the other. In some respects this case is easier to analyze. Conjecture 3, for instance, holds.

(3) The *directed/undirected variation.* When a chromosome is analyzed, sometimes only the adjacency of genes is determined, not their absolute order. In the linear case this means the given order may be left to right or right to left, and in the circular case, clockwise or counterclockwise.

(4) The *weighted/unweighted variation.* We have assumed that all reversals are equally probable, so it is appropriate to count their number. For a first approximation this is reasonable, but a finer analysis might weight reversals by a function of their length, and find a series of minimum total weight.

(5) The *pairwise/multiple variation.* Though solution of the reversal distance problem does not yield an alignment of the input, one can define a multiple chromosome comparison problem in analogy to multiple sequence alignment. In particular, given a collection of permutations of the same genes, how can one infer an evolutionary tree and a set of hypothetical ancestral permutations so as to minimize the total reversal distance along edges of the tree? Practical exact solution appears unlikely.

Clearly there are many avenues for exploration.

Acknowledgements

Guillaume Leduc, in analyzing signed reversals, contributed to the line of proof that $\Phi(\pi)$ bounds $\mathrm{Opt}(\pi)$. Holger Gollan began the study of the distribution of $\mathrm{Opt}(\pi)$ for small n, and with his conjectures on the diameter of the symmetric group generated by reversals, stimulated the search for better lower bounds. Michel Berkelaar generously provided the sparse linear programming code used in our experiments.

Our thanks to the referees for their comments. The first author also wishes to thank Paul Stelling and Lucas Hui for the knot decomposition counterexample, and Dan Gusfield for many valuable discussions.

References

[1] Aigner, Martin and Douglas B. West. Sorting by insertion of leading elements. *Journal of Combinatorial Theory* (Series A) 45, 306–309, 1987.

[2] Amato, Nancy, Manuel Blum, Sandra Irani, and Ronitt Rubinfeld. Reversing trains: a turn of the century sorting problem. *Journal of Algorithms* 10, 413–428, 1989.

[3] Bibb, J.J., R.A. van Etten, C.T. Wright, M.W. Walberg, and D.A. Clayton. *Cell* 26, 167–180, 1981.

[4] Dobzhansky, Theodosius. *Genetics of the Evolutionary Process.* Columbia University Press, 1970.

[5] Driscoll, James R. and Merrick L. Furst. Computing short generator sequences. *Information and Computation* 72, 117–132, 1987.

[6] Even, S. and O. Goldreich. The minimum-length generator sequence problem is NP-hard. *Journal of Algorithms* 2, 311–313, 1981.

[7] Furst, Merrick, John Hopcroft, and Eugene Luks. Polynomial-time algorithms for permutation groups. In *Proceedings of the 21st Symposium on Foundations of Computer Science*, 36–41, 1980.

[8] Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to The Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[9] Gates, William H. and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics* 27, 47–57, 1979.

[10] Gollan, Holger. Personal communication, 1991.

[11] Jerrum, Mark R. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science* 36, 265–289, 1985.

[12] Johnson, Donald B. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing* 4:1, 77–84, 1975.

[13] Knuth, Donald E. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.

[14] Mannila, Heikki. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers* 34, 318–325, 1985.

[15] Micali, S. and V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matchings in general graphs. In *Proceedings of the 21st Symposium on Foundations of Computer Science*, 17–27, 1980.

[16] Nadeau, J.H. and B.A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proceedings of the National Academy of Sciences USA* 81, 814, 1984.

[17] O'Brien, S.J., editor. *Genetic Maps.* Cold Spring Harbor Laboratory, 1987.

[18] Palmer, J.D., B. Osorio, and W.F. Thompson. Evolutionary significance of inversions in legume chloroplast DNAs. *Current Genetics* 14, 65–74, 1988.

[19] Sankoff, David, Guillame Leduc, Natalie Antoine, Bruno Paquin, B. Franz Lang, and Robert Cedergren. Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome. *Proceedings of the National Academy of Sciences USA* 89, 6575–6579, 1992.

[20] Schöniger, Michael and Michael S. Waterman. A local algorithm for DNA sequence alignment with inversions. *Bulletin of Mathematical Biology* 54, 521–536, 1992.

[21] Sessions, Stanley K. Chromosomes: molecular cytogenetics. In *Molecular Systematics*, David M. Hillis and Craig Moritz editors, Sinauer, Sunderland, Massachusetts, 156–204, 1990.

[22] Tichy, Walter F. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems* 2:4, 309–321, 1984.

[23] Wagner, Robert A. On the complexity of the extended string-to-string correction problem. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, David Sankoff and Joseph B. Kruskal, editors, Addison-Wesley, Reading Massachusetts, 215–235, 1983.

[24] Watterson, G.A., W.J. Ewens, T.E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology* 99, 1–7, 1982.

[25] Wolstenholme, D.R., J.L. MacFarlane, R. Okimoto, D.O. Clary, and J.A. Wahleithner. *Proceedings of the National Academy of Sciences USA* 84, 1324–1328, 1987.