A Linear-Time Algorithm for Computing Inversion Distance Between Signed Permutations with an Experimental Study

David A. Bader[1,⋆], Bernard M.E. Moret[2,⋆⋆], and Mi Yan[1,⋆⋆⋆]

[1] Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA. {dbader,miyan}@eece.unm.edu

[2] Department of Computer Science, University of New Mexico, Albuquerque, NM 87131 USA. moret@cs.unm.edu

Abstract. Hannenhalli and Pevzner gave the first polynomial-time algorithm for computing the inversion distance between two signed permutations, as part of the larger task of determining the shortest sequence of inversions needed to transform one permutation into the other. Their algorithm (restricted to distance calculation) proceeds in two stages: in the first stage, the overlap graph induced by the permutation is decomposed into connected components, then in the second stage certain graph structures (hurdles and others) are identified. Berman and Hannenhalli avoided the explicit computation of the overlap graph and gave an $O(n\alpha(n))$ algorithm, based on a Union-Find structure, to find its connected components, where α is the inverse Ackerman function. Since for all practical purposes $\alpha(n)$ is a constant no larger than four, this algorithm has been the fastest practical algorithm to date. In this paper, we present a new linear-time algorithm for computing the connected components, which is more efficient than that of Berman and Hannenhalli in both theory and practice. Our algorithm uses only a stack and is very easy to implement. We give the results of computational experiments over a large range of permutation pairs produced through simulated evolution; our experiments show a speed-up by a factor of 2 to 5 in the computation of the connected components and by a factor of 1.3 to 2 in the overall distance computation.

1 Introduction

Some organisms have a single chromosome or contain single-chromosome organelles (such as mitochondria or chloroplasts), the evolution of which is largely independent on the evolution of the nuclear genome. Given a particular strand from a single chromosome, whether linear or circular, we can infer the ordering and directionality of the genes, thus representing each chromosome by an ordering of oriented genes. In many cases, the evolutionary process that operates on such single-chromosome organisms consists mostly of inversions of portions of the chromosome; this finding has led many

⋆ Supported in part by NSF Grants CAREER 00-93039, NSF ITR 00-81404 and NSF DEB 99-10123 and by DOE SUNAPP AX-3006 and DOE-CSRI-14968.

⋆⋆ Supported in part by NSF Grant ITR 00-81404.

⋆⋆⋆ Supported by the UNM Albuquerque High Performance Computing Center.

biologists to reconstruct phylogenies based on gene orders, using as a measure of evolutionary distance between two genomes the inversion distance, i.e., the smallest number of inversions needed to transform one signed permutation into the other [18, 19, 20].

Both inversion distance and the closely related transposition distance are difficult computational problems that have been studied intensively over the last five years [1, 3, 4, 5, 7, 8, 11, 12]. Finding the inversion distance between unsigned permutations is NP-hard [7], but with signed ones, it can be done in polynomial time [11]. The fastest published algorithm for the computation of inversion distance between two signed permutations has been that of Berman and Hannenhalli [5], which uses a Union-Find data structure and runs in $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function. (The later KST algorithm [12] reduces the time needed to compute the shortest sequence of inversions, but uses the same algorithm for computing the length of that sequence.) We have found only two implementations on the web, both designed to compute the shortest sequence of inversions as well as its length; one, due to Hannenhalli [10], implements his first algorithm [11], which runs in quadratic time when computing distances, while the other, a Java applet written by Mantin [13], a student of Shamir, implements the KST algorithm [12], but uses an explicit representation of the overlap graph and thus also takes quadratic time.

We present a simple and practical, worst-case linear-time algorithm to compute the connected components of the overlap graph, which results in a simple linear-time algorithm for computing the inversion distance between two signed permutations. We also provide ample experimental evidence that our linear-time algorithm is efficient in practice as well as in theory: we coded it as well as the algorithm of Berman and Hannenhalli, using the best principles of algorithm engineering [14, 16] to ensure that both implementations would be as efficient as possible, and compared their running times on a large range of instances generated through simulated evolution. (The two implemenations on the web are naturally far slower.)

The paper is organized as follows. We begin by recalling some definitions, briefly review past work on sorting by reversals, then introduce the concepts that we will need in our algorithm, including the fundamental theorem that makes it possible. We then describe and analyze our algorithm, discuss our experimental setup, present and comment on our results, and briefly mention an application of our distance computation in a whole-genome phylogeny study.

2 Inversions on Signed Permutations

We assume a fixed set of genes $\{g_1, g_2, \ldots, g_n\}$. Each genome is then an ordering (circular or linear) of these genes, each gene given with an orientation that is either positive (g_i) or negative ($-g_i$). The ordering $g_1, g_2, \ldots, g_n$, whether linear or circular, is considered equivalent to that obtained by considering the complementary strand, i.e., the ordering $-g_n, -g_{n-1}, \ldots, -g_1$.

Let G be the genome with signed ordering (linear or circular) $g_1, g_2, \ldots, g_n$. An *inversion* between indices i and j, for $i \leq j$, produces the genome with linear ordering

$$g_1, g_2, \ldots, g_{i-1}, -g_j, -g_{j-1}, \ldots, -g_i, g_{j+1}, \ldots, g_n$$

If we have $j < i$, we can still apply an inversion to a circular (but not linear) genome by rotating the circular ordering until the two indices are in the proper relationship—recall that we consider all rotations of the complete circular ordering of a circular genome as equivalent.

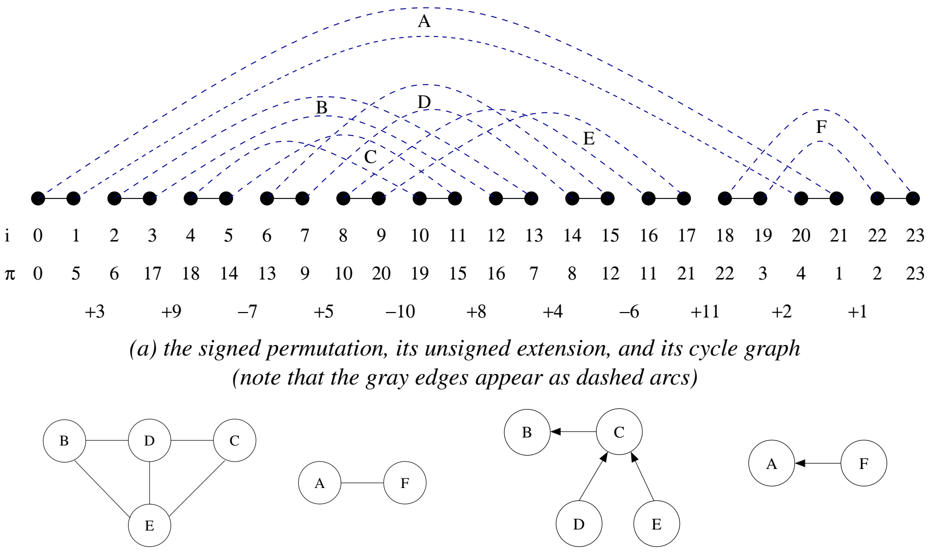
The inversion distance between two genomes (two signed permutations of the same set) is then the minimum number of inversions that must be applied to one genome in order to produce the other. (This measure is easily seen to be a true metric.) Computing the shortest sequence of inversions that gives rise to this distance is also known as *sorting by reversals*—we shall shortly see why it can be regarded as a sorting problem.

3 Previous Work

Bafna and Pevzner introduced the cycle graph of a permutation [2], thereby providing the basic data structure for inversion distance computations. Hannenhalli and Pevzner then developed the basic theory for expressing the inversion distance in easily computable terms (number of breakpoints minus number of cycles plus number of hurdles plus a correction factor for a fortress [2, 22]—hurdles and fortresses are easily detectable from a connected component analysis). They also gave the first polynomial-time algorithm for sorting signed permutations by reversals [11]; they also proposed a $O(n^4)$ implementation of their algorithm [10], which runs in quadratic time when restricted to distance computation. Their algorithm requires the computation of the connected components of the overlap graph, which is the bottleneck for the distance computation. Berman and Hannenhalli later exploited some combinatorial properties of the cycle graph to give a $O(n\alpha(n))$ algorithm to compute the connected components, leading to a $O(n^2\alpha(n))$ implementation of the sorting algorithm [5]. (We will refer to this approach as the UF approach.) Algorithms for finding the connected components of interval graphs (a class of graphs that include the more specialized overlap graphs used in sorting by reversals) that run in linear time are known, but they use range minima and lowest common ancestor data structures and algorithms so that, in addition to being complex and hard to implement, they suffer from high overhead—high enough, in fact, that the UF approach would remain the faster solution in practice.

4 Overlap Graph and Forest

Given a signed permutation of $\{1,\ldots,n\}$, we transform it into an unsigned permutation π of $\{1,\ldots,2n\}$ by substituting the ordered pair $(2x-1,2x)$ for the positive element x and the ordered pair $(2x,2x-1)$ for the negative elements $-x$, then extend π to the set $\{0,1,\ldots,2n,2n+1\}$ by setting $\pi(0) = 0$ and $\pi(2n+1) = 2n+1$. By convention, we assume that the two signed permutations for which we must compute a distance have been turned in this manner into unsigned permutations and then both permuted (by the same transformation) so that the first permutation becomes the linear ordering $(0,1,\ldots,2n,2n+1)$; these manipulations do not affect the distance value. (This is the reason why transforming one permutation into the other can be viewed as sorting—we want to find out how many inversions are needed to produce the identity permutation from the given one.) We represent an extended unsigned permutation with an edge-colored graph, the *cycle graph* of the permutation. The graph has $2n+2$ vertices; for each i, $0 \le i \le n$, we join vertices $\pi(2i)$ and $\pi(2i+1)$ by a *gray* edge and vertices $2i$ and $2i+1$ by a *black* edge, as illustrated in Figure 1(a). The resulting graph consists of disjoint cycles in which edges alternate colors; we remove from it all 2-cycles (because these cycles correspond to portions of the permutation that are already sorted and cannot intersect with any other cycles). We say that gray edges $(\pi(i), \pi(j))$ and $(\pi(k), \pi(t))$



(a) the signed permutation, its unsigned extension, and its cycle graph (note that the gray edges appear as dashed arcs)

(b) the overlap graph *(c) the overlap forest*

Fig. 1. The signed permutation $(+3, +9, -7, +5, -10, +8, +4, -6, +11, +2, +1)$ and its various representations

overlap whenever the two intervals $[i, j]$ and $[k, t]$ overlap, but neither contains the other. Similarly, we say that cycles C_1 and C_2 *overlap* if there exist overlapping gray edges $e_1 \in C_1$ and $e_2 \in C_2$.

Definition 1. *The* overlap graph *of permutation* π *has one vertex for each cycle in the cycle graph and an edge between any two vertices that correspond to overlapping cycles.*

Figure 1 illustrates the concept. The *extent* of cycle C is the interval $[C.B, C.E]$, where we have $C.B = \min\{i \mid \pi(i) \in C\}$ and $C.E = \max\{i \mid \pi(i) \in C\}$. The extent of a set of cycles $\{C_1, \ldots, C_k\}$ is $[B, E]$, with $B = \min_{i=1}^{k} C_i.B$ and $E = \max_{i=1}^{k} C_i.E$. In Figure 1, the extent of cycle A is [0,21], that of cycle F is [18,23], and that of the set $\{A, F\}$ is [0,23].

No algorithm that actually builds the overlap graph can run in linear time, since that graph can be of quadratic size. Thus, our goal is to construct an *overlap forest* such that two vertices f and g belong to the same tree in the forest exactly when they belong to the same connected component in the overlap graph. An overlap forest (the composition of its trees is unique, but their structure is arbitrary) has exactly one tree per connected component of the overlap graph and is thus of linear size.

5 The Linear-Time Algorithm for Connected Components

Our algorithm for computing the connected components scans the permutation twice. The first scan sets up a trivial forest in which each node is its own tree, labelled with the beginning of its cycle. The second scan carries out an iterative refinement of this first forest, by adding edges and so merging trees in the forest; unlike a Union-Find, however, our algorithm does not attempt to maintain the trees within certain shape parameters.

Recall that a node in the overlap graph (or forest) corresponds to a cycle in the cycle graph. The extent $[f.B, f.E]$ of a node f of the overlap forest is the extent of the set of nodes in the subtree rooted at f. Let F_0 be the trivial forest set up in the first scan and assume that the algorithm has processed elements 0 through $j-1$ of the permutation, producing forest F_{j-1}. We construct F_j from F_{j-1} as follows. Let f be the cycle containing element j of the permutation. If j is the beginning of its own cycle f, then it must be the root of a single-node tree; otherwise, if f overlaps with another cycle g, then we add a new arc (g, f) and compute the combined extent of g and of the tree rooted at f. We say that a tree rooted at f is *active* at stage j whenever j lies properly within the extent of f; we shall store the extent of the active trees in a stack.

Figure 2 summarizes our algorithm for constructing the overlap forest; in the algorithm, *top* denotes the top element of the stack. The conversion of a forest of up-trees into connected component labels is accomplished in linear time by a simple sweep of the array, taking advantage of the fact that the parent of i must appear before i in the array.

Lemma 1. *At iteration i of Step (3) of the algorithm, if the tree rooted at* top *is active and i lies on cycle f and we have $f.B <$* top*.B, then there exists h in the tree rooted at* top *such that h overlaps with f.*

Proof. Since *top* is active, it must have been pushed onto the stack before the current iteration ($top.B < i$) and we must not have reached the end of *top*'s extent ($i < top.E$). Hence, i must be contained in *top*'s extent ($top.B < i < top.E$). Since i lies on the cycle f that begins before *top* ($f.B < top.B$), there must be an edge from cycle f that overlaps with *top*.

Theorem 1. *The algorithm produces a forest in which each tree is composed of exactly those nodes that form a connected component.*

Proof. It suffices to show that, after each iteration, the trees in the forest correspond exactly to the connected components determined by the permutation values scanned up to that point. We prove this invariant by induction on the number of applications of Step (3) of the algorithm.

The base case is trivial: each tree of F_0 has a single node and no two nodes belong to the same connected component since we have not yet processed any element of the permutation.

Input: permutation
Output: *parent*$[i]$, the parent of i in the overlap forest
Begin

1. scan the permutation, label each position i with $C[i].B$, and set up $[C[i].B, C[i].E]$
2. initialize empty stack
3. for $i \leftarrow 0$ to $2n+1$
   (a) if $i = C[i].B$
   then push $C[i]$
   (b) *extent* $\leftarrow C[i]$
   while $(\mathit{top}.B > C[i].B)$
   $\mathit{extent}.B \leftarrow \min\{\mathit{extent}.B, \mathit{top}.B\}$
   $\mathit{extent}.E \leftarrow \max\{\mathit{extent}.E, \mathit{top}.E\}$
   pop *top*
   $\mathit{parent}[\mathit{top}.B] \leftarrow C[i].B$
   endwhile
   $\mathit{top}.B \leftarrow \min\{\mathit{extent}.B, \mathit{top}.B\}$
   $\mathit{top}.E \leftarrow \max\{\mathit{extent}.E, \mathit{top}.E\}$
   (c) if $i = \mathit{top}.E$
   then pop *top*
4. convert each tree into a labeling of its vertices

End

Fig. 2. Constructing the Interleaving Forest in Linear Time

Assume that the invariant holds after the $(i-1)$st iteration and let i lie on cycle f. We prove that the nodes of the tree containing i form the same set as the nodes of the connected component containing i—other trees and connected components are unaffected and so still obey the invariant.

- We prove that a node in the tree containing i must be in the same connected component as i.
  If we have $i = f.B$, then, as we remarked earlier, nothing changes in the overlap graph (and thus in the connected components); from Step (3), it is also clear that the forest remains unchanged, so that the invariant is preserved.
  On the other hand, if we have $i > f.B$, then at Step (3) the edge (top, f) will be added to the forest whenever $f.B < top.B$ holds. This edge will join the subtree rooted at f with that rooted at *top* into a single subtree. From Lemma 1, we also know that, whenever $f.B < top.B$ holds, there must exist h in the tree rooted at *top* such that h and f overlap, so that edge (h, f) must belong to the overlap graph, thereby connecting the component containing f with that containing *top* and merging them into a single connected component, which maintains the invariant.
- We prove that a node in the same connected component as i must be in the tree containing i. Whenever (j,i) and (k,l), with $j < k < i < l$, are gray edges on cycles f and h respectively, then edge (f,h) must belong to the overlap graph built from the first i entries of the permutation. In such a case, our algorithm ensures that edge (h, f) belongs the overlap forest. Our conclusion follows.

Obviously, each step of the algorithm takes linear time, so the entire algorithm runs in worst-case linear time.

6 Experiments

6.1 Programs

The implementation due to Hannenhalli is very slow and implements the original method of Hannenhalli and Pevzner and not the faster one of Berman and Hannenhalli. The KST applet is very slow as well since it explicitly constructs the overlap graph; it is also written in Java which makes it difficult to compare with C code. For these reasons we wrote our own implementation of the Berman and Hannenhalli algorithm (just the part handling the distance computation) with a view to efficiency. Thus, we not only have an efficient implementation to compare to our linear-time algorithm, but also we have ensured that the two implementations are truly comparable because they share much of their code (hurdles, fortresses, breakpoints), were written by the same person, and used the same algorithmic engineering techniques.

6.2 Experimental Setup

We ran experiments on signed permutations of length 10, 20, 40, 80, 160, 320, and 640, in order to verify rate of growth as a function of the number of genes and also to cover the full range of biological applications. We generated groups of 3 signed permutations from the identity permutation using the evolutionary model of Nadeau and Taylor [17]; in this model, randomly chosen inversions are applied to the permutation at a node to generate the permutations labelling its children, repeating the process until all nodes have been assigned a permutation. The expected number of inversions per edge, r, is fixed in advance, reflecting assumptions about the evolutionary rate in the model. We use 5 evolutionary rates: 4, 16, 64, 256, and 1024 inversions per edge and generated 10 groups of 3-leaf trees—or 10 groups of 3 genomes each—at each of the 6 selected lengths. We also generated 10 groups of 3 random permutations (from a uniform distribution) at each length to provide an extreme test case. For each of these 36 test suites, we computed the 3 distances among the 3 genomes in each group 20,000 times in a tight loop, in order to provide accurate timing values for a single computation, then averaged the values over the 10 groups and computed the standard deviation. The computed inversion distances are expected to be at most twice the evolutionary rate since there are two tree edges between each pair of genomes. Our linear algorithm exhibited very consistent behavior throughout, with standard deviations never exceeding 2% of the mean; the UF algorithm showed more variation for $r = 4$ and $r = 16$.

We ran all our tests on a 300MHz Pentium II with 16KB of L1 data cache, 16KB of L1 instruction cache, and 512KB of L2 cache running at half clockspeed; our codes were compiled under Linux with the GNU gcc compiler with options `-O3 -mpentiumpro`. Our code also runs on other systems and machines (e.g., Solaris and Microsoft), where we observed the same behavior.

6.3 Experimental Results

We present our results in four plots. The first two (Figure 4) show the actual running time of our linear-time algorithm for the computation of the inversion distance between

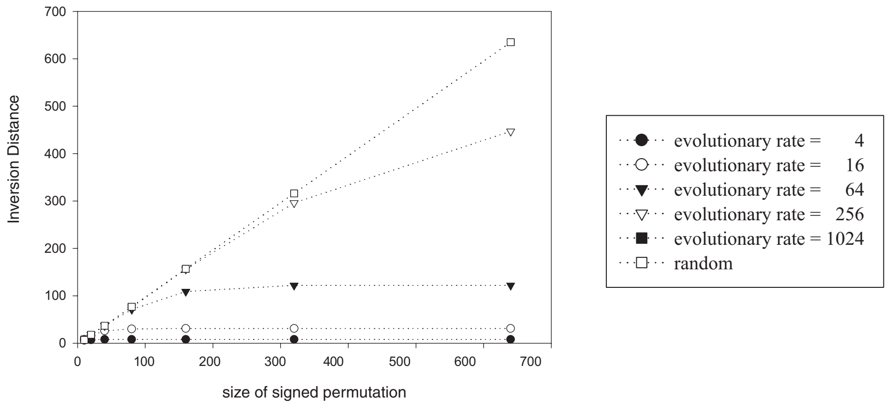


Fig. 3. The inversion distance as a function of the size of the signed permutation.

two permutations as a function of the size of the permutation, with one plot for the computation of the connected components alone and the other for the complete distance computation. Each plot shows one curve each for the various evolutionary rates and one for the random permutations. We added a third plot showing the average inversion distance; note the very close correlation between the distance and the running time.

For small permutation sizes (10 or less), the L1 data cache holds all of the data without any cache misses, but, as the permutation size grows, the hit rate in the direct-mapped L1 cache steadily decreases until, for permutations of size 100 and larger, execution has slowed down to the speed of the L2 cache (a ratio of 2). From that point on, it is clear that the rate of growth is linear, as predicted. It is also clear that $r = 1024$ is as high a rate of evolution as we need to test, since the number of connected components and inversion distance are nearly indistinguishable from those of the random permutations (see the plot in Figure 3 that plots inversion distance as a function of the permutation size). The speed is remarkable: for a typical genome of 100 gene fragments (as found in chloroplast data, for instance [9]), well over 20,000 distance computations can be carried out every second on our rather slow workstation.

Our second two plots (Figure 5) compare the speed of our linear-time algorithm and that of the UF approach. We plot speedup ratios, i.e., the ratio of the running time of the UF approach to that of our linear-time algorithm. Again, the first plot addresses only the connected components part of the computation, while the second captures the complete distance computation.

Since the two approaches use a different amount of memory and give rise to a very different pattern of addressing (and thus different cache conflicts), the ratios up to permutations of size 100 vary quite a bit as the size increases—reflecting a transition from the speed of the L1 cache to that of the L2 cache at different permutation sizes for the two algorithms. Beyond that point, however, the ratios stabilize and clearly demonstrate the gain of our algorithm, a gain that increases with increasing permutation size as well as with increasing evolutionary rate.

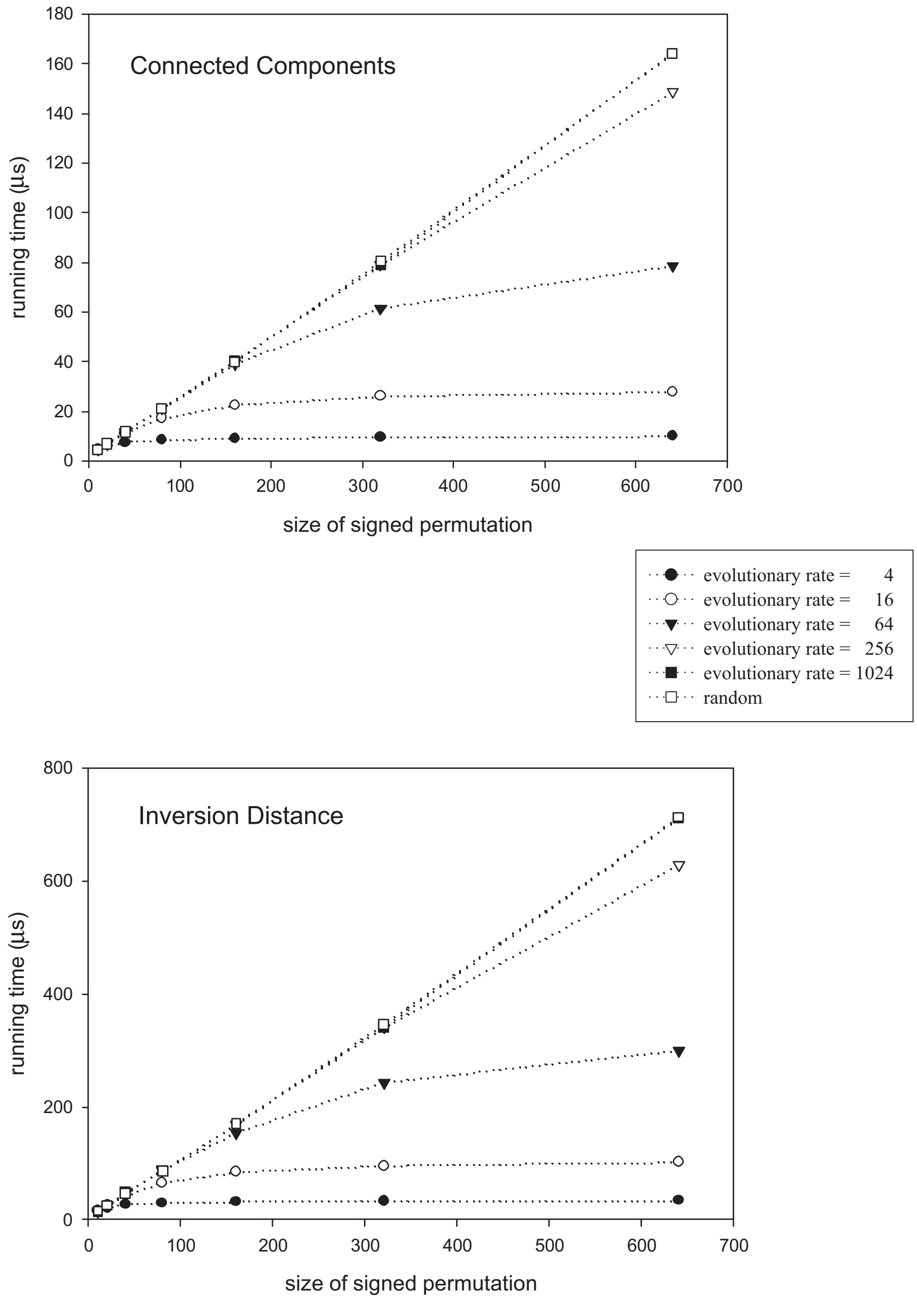


Fig. 4. The running time of our linear-time algorithms as a function of the size of the signed permutation.

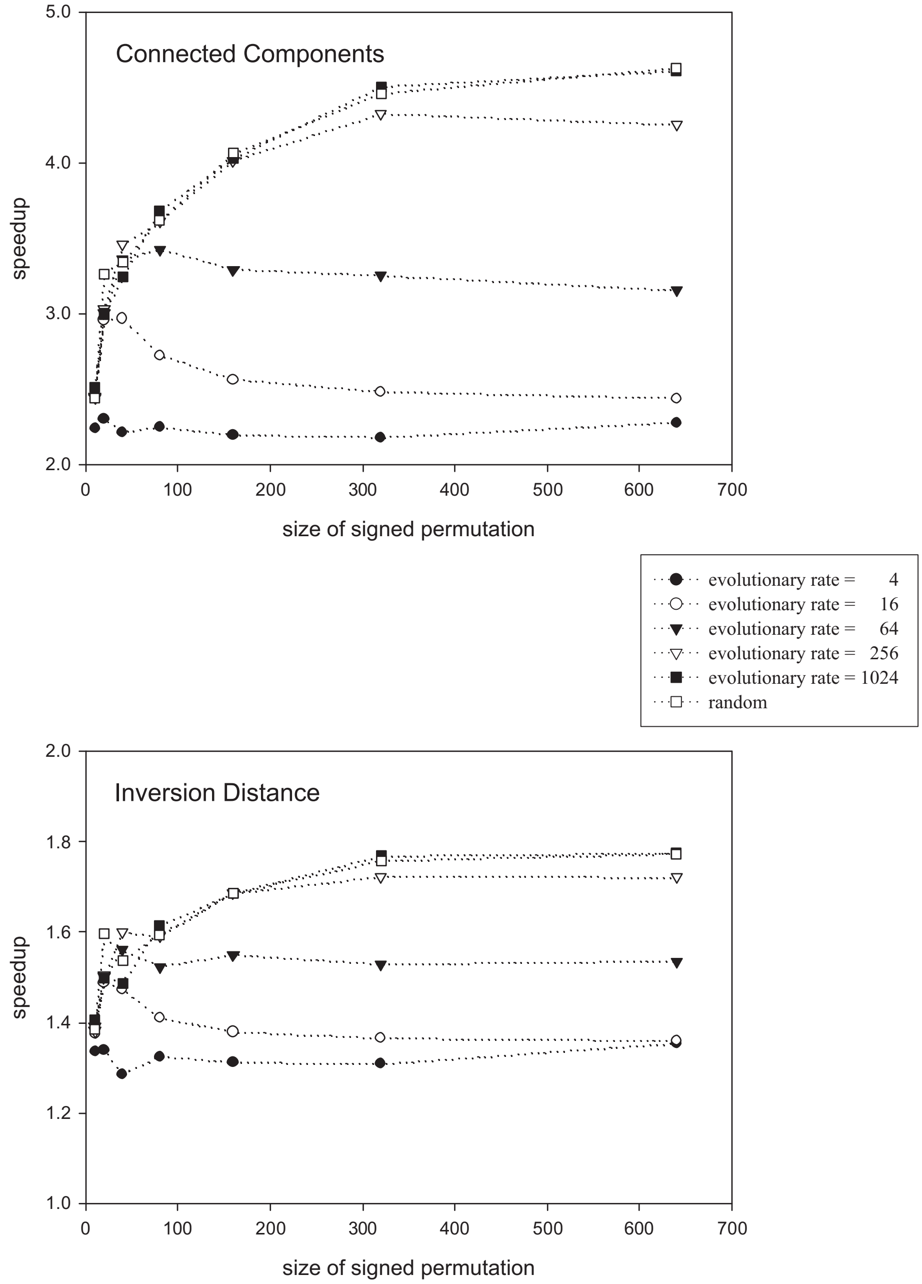


Fig. 5. The speedup of our linear-time algorithms over the UF approach as a function of the size of the signed permutation.

7 Concluding Remarks

We have presented a new, very simple, practical, linear-time algorithm for computing the inversion distance between two signed permutations, along with a detailed experimental study comparing the running time of our algorithm with that of Berman and Hannenhalli. Our code is available from the web page `www.cs.unm.edu/~moret/GRAPPA` under the terms of the GNU Public License (GPL); it has been tested under Linux (including the parallel version), FreeBSD, Solaris, and Windows NT. This code includes inversion distance as part of a much larger context, which provides means of reconstructing phylogenies based on gene order data. We found that using our inversion distance computation in lieu of the surrogate breakpoint distance (which was used by previous researchers in an attempt to speed up computation [6, 21]) only slowed down the reconstruction algorithm by about 30%, enabling us to extend work on breakpoint analysis (as reported in [9, 15]) to similar work on inversion phylogeny.

References

[1] V. Bafna and P. Pevzner. Sorting permutations by transpositions. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 614–623, New York, January 1995. ACM Press.

[2] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS93)*, pages 148–157. IEEE Press, 1993.

[3] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25:272–289, 1996.

[4] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11:224–240, 1998.

[5] P. Berman and S. Hannenhalli. Fast Sorting by Reversal. In D.S. Hirschberg and E.W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 168–185, Laguna Beach, CA, June 1996. Lecture Notes in Computer Science, **1075**, Springer-Verlag.

[6] M. Blanchette, G. Bourque, and D. Sankoff. Breakpoint phylogenies. In S. Miyano and T. Takagi, editors, *Genome Informatics*, pages 25–34. University Academy Press, Tokyo, Japan, 1997.

[7] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the 1st Conference on Computational Molecular Biology (RECOMB97)*, pages 75–83, Santa Fe, NM, 1997. ACM Press.

[8] A. Caprara. Sorting permutations by reversals and Eulerian cycle decompositions. *SIAM J. Discrete Math.*, 12(1):91–110, 1999.

[9] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.-S. Wang, T. Warnow, and S. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB00)*, pages 104–115, San Diego, CA, 2000.

[10] S. Hannenhalli. *Software for computing inversion distances between signed gene orders*. Department of Mathematics, University of Southern California, URL. `www-hto.usc.edu/plain/people/Hannenhalli.html`.

[11] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th Annual Symposium on Theory of Computing (STOC95)*, pages 178–189, Las Vegas, NV, 1995. ACM Press.

[12] H. Kaplan, R. Shamir, and R.E. Tarjan. A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal of Computing*, 29(3):880–892, 1999. First appeared in *Proceedings of the 8th Annual Symposium on Discrete Algorithms (SODA97)*, 344-351, New Orleans, LA. ACM Press.

[13] I. Mantin and R. Shamir. Genome Rearrangement Algorithm Applet: An algorithm for sorting signed permutations by reversals. `www.math.tau.ac.il/~rshamir/GR/`, 1999.

[14] C.C. McGeoch. Toward an experimental method for algorithm simulation. *INFORMS Journal of Computing*, 8:1–15, 1996.

[15] B. M.E. Moret, S. Wyman, D.A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proceedings of the 6th Pacific Symposium on Biocomputing (PSB2001)*, pages 583–594, Big Island, HI, January 2001.

[16] B.M.E. Moret. Towards a discipline of experimental algorithmics. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 2001. To appear. Available at `www.cs.unm.edu/~moret/dimacs.ps`.

[17] J.H. Nadeau and B.A. Taylor. Lengths of chromosome segments conserved since divergence of man and mouse. In *Proceedings of the National Academy of Sciences*, volume 81, pages 814–818, 1984.

[18] R.G. Olmstead and J.D. Palmer. Chloroplast DNA systematics: a review of methods and data analysis. *American Journal of Botany*, 81:1205–1224, 1994.

[19] J.D. Palmer. Chloroplast and mitochondrial genome evolution in land plants. In R. Herrmann, editor, *Cell Organelles*, pages 99–133. Springer Verlag, 1992.

[20] L.A. Raubeson and R.K. Jansen. Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants. *Science*, 255:1697–1699, 1992.

[21] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5:555–570, 1998.

[22] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS, Boston, MA, 1997.