Aligning Genomes with Inversions and Swaps

J. L. Holloway
Crop and Soil Science
Oregon State University
holloway@bcc.orst.edu

P. Cull
Computer Science
Oregon State University
pc@cs.orst.edu

Abstract

The decision about what operators to allow and how to charge for these operations when aligning strings that arise in a biological context is the decision about what model of evolution to assume. Frequently the operators used to construct an alignment between biological sequences are limited to deletion, insertion, or replacement of a character or block of characters, but there is biological evidence for the evolutionary operations of exchanging the positions of two segments in a sequence and the replacement of a segment by its reversed complement.

In this paper we describe a family of heuristics designed to compute alignments of biological sequences assuming a model of evolution with swaps and inversions. The heuristics will necessarily be approximate since the appropriate way to charge for the evolutionary events (delete, insert, substitute, swap, and invert) is not known. The paper concludes with a pairwise comparison of 20 Picornavirus genomes, and a detailed comparison of the hepatitis delta virus with the citrus exocortis viroid.

The Problem

Given two strings P and T with $|P| \leq |T|$, we want to find an alignment $\mathcal{X}$ of P within T. An alignment $\mathcal{X}$ is a function mapping $[1,|P|]$ into $[1,|T|]$. Given a number in $[1,|P|]$, $\mathcal{X}$ will return a number in $[1,|T|]$ indicating the character in T with which the character in P is aligned. To decide when one alignment is better than another, we need a function, $\mathcal{F}$, to evaluate an alignment $\mathcal{X}$ given P and T. The string alignment problem is: given $\mathcal{F}$, P, and T, find an alignment $\mathcal{X}$ that maximizes $\mathcal{F}(P,T,\mathcal{X})$. The decision version of the problem is: given $\mathcal{F}$, P, T and an integer L, is there an alignment $\mathcal{X}$ so that $\mathcal{F}(P,T,\mathcal{X}) \geq L$?

If $\mathcal{F}$ can be evaluated in polynomial time, then the decision version is in $\mathcal{NP}$ since one merely has to guess $\mathcal{X}$ and evaluate. Further it is trivial to show that if $\mathcal{F}$ is not further restricted, then the problem is $\mathcal{NP}$–complete.

To create an $\mathcal{F}$ that captures some biological intuitions, we consider $\mathcal{F}$ to be made of two parts. The first part should give a positive contribution based on the similarity between aligned characters. The second part should give a negative contribution that depends on the number and length of gaps, swaps and inversions. A gap in an alignment occurs when adjacent characters in the pattern are aligned with non–adjacent characters in the text. The length of the gap is the number of characters between the non–adjacent characters in the text. A swap occurs when two substrings, $P_1 = p_i p_{i+1} \cdots p_j$ and $P_2 = p_k p_{k+1} \cdots p_l$, $j \leq k$, are matched with text so that the text matched with P_2 occurs to the left of the text matched with P_1. An inversion occurs when a substring, $P_1 = p_i p_{i+1} \cdots p_j$, is matched with text that has the form $\overline{p_j p_{j-1}} \cdots \overline{p_i}$. We use $\overline{p_i}$ to indicate the complement of p_i.

The first part of $\mathcal{F}$ could have the form $\sum_{i=1}^{|P|} \mathcal{M}(P_i, T_{\mathcal{X}(i)})$ where $\mathcal{M}$ takes two characters, the i^{th} character from P and the corresponding aligned character from T, and returns an integer indicating how similar the characters are. In general, we allow $\mathcal{M}$ to take its two characters from each of two alphabets, and allow $\mathcal{M}$ to give negative values when certain characters should not be aligned. In nucleic acid applications, $\mathcal{M}$ could return a positive quantity, c, when the aligned nucleotides are identical, and return 0 when the aligned nucleotides are not identical. For protein applications, $\mathcal{M}$ could do a look–up in an appropriate PAM matrix.

For the second part of $\mathcal{F}$ we expect a more negative contribution as the size of the gap, swap or inversion gets larger. We can use $|\mathcal{X}(i+1) - \mathcal{X}(i) - 1|$ as the measure of gap size. To penalize gaps, we introduce a gap function $\mathcal{G}$ which is an increasing function of the gap size and has the property that $\mathcal{G}(0) = 0$. The gap penalty is

$$-\sum_{i=1}^{|P|-1} \mathcal{G}(|\mathcal{X}(i+1) - \mathcal{X}(i) - 1|)$$

In the simplest case, we may take $\mathcal{G}(x) = ax$. Notice that this form of the gap penalty allows swaps and penalizes a swap about as much as a gap. We can introduce a similar function $\mathcal{I}$ to penalize inversions differently than gaps and swaps. For now, we assume

that gaps, swaps, and inversions are penalized with the function $\mathcal{G}$.

The problem now is given P and T

maximize

$$\left(\sum_{i=1}^{|P|} \mathcal{M}(P_i, T_{\mathcal{X}(i)}) - \sum_{i=1}^{|P|-1} \mathcal{G}(|\mathcal{X}(i+1) - \mathcal{X}(i) - 1|)\right)$$

In the simplest case this becomes

maximize

$$\left(c\sum_{i=1}^{|P|} EQ(P_i, T_{\mathcal{X}(i)}) - a\sum_{i=1}^{|P|-1} |\mathcal{X}(i+1) - \mathcal{X}(i) - 1|\right)$$

where EQ returns 1 when its arguments are equal and returns 0 otherwise.

Previous Work on Related Problems

Solutions to the edit distance problem, given two strings, find the fewest edit operations required to make the two strings identical, have been given by several researchers. The operations allowed are, delete a single character from either string and change a single character in one string to match the corresponding character in the other string. Reviews of the edit distance problem and similar problems solved using dynamic programming can be found in (Goad 1986; Waterman 1984; 1988). Several recent algorithms charge differently for deleting a block. In 1988 Miller & Myers (Miller & Myers 1988), in 1989 Galil & Giancarlo (Galil & Giancarlo 1989), and in 1990 Eppstein, Galil, & Giancarlo (Eppstein, Galil, & Giancarlo 1990) developed several algorithms to improve the speed of these dynamic programming algorithms.

The dynamic programming algorithms all require that the order of the characters be the same in the pattern and the text. The model of genome shuffling proposed by Sankoff and Goldstein (Sankoff & Goldstein 1989) suggests that the algorithms used to align genetic sequences should not be limited to those that maintain the order of the characters in the strings. Several people have proposed algorithms to align strings while not strictly requiring that all matched characters be in the same order in the pattern and text. Lowrance & Wagner (Lowrance & Wagner 1975) allow the operation of swapping two adjacent characters in one of the strings. In 1990, A. Bertossi, E. Lodi, F. Luccio, & L. Pagli (Bertossi *et al.* 1990) considered context dependent approximate string matching and added the operation of transposing two characters. In 1992, M. Schoniger & M. Waterman (Schoniger & Waterman 1992) gave a dynamic programming algorithm that finds local alignments with inversions although the algorithm does not find inverted segments within other inverted segment. Tichy (Tichy 1984) gave an algorithm that converts one string to another using only block move operations. This algorithm minimizes the number of block moves, but does not consider the size of the blocks being moved.

Heuristic

Our heuristic is designed using a combination of the dynamic programming and divide and conquer methods. As in dynamic programming, the heuristic tries matching the pattern within the first $l+1$ characters of the text by comparing the score found using the $l+1^{st}$ character with the score found by matching the pattern within the first l characters of the text. The divide and conquer flavor appears in the way the score at position l is computed. The heuristic divides the pattern string into two half size strings, and aligns the first half and the second half so that the sum of the scores for these two alignments minus a gap penalty is maximized. As usual, the divide and conquer process would give rise to a binary tree for the recursive calls. Although a top–down description of a divide and conquer algorithm is elegant, the overhead for recursive subroutine calls usually makes a bottom–up algorithm more efficient in practice. The dynamic programming philosophy suggests saving the results of intermediate computations, so that if these results are needed again they will not need to be recomputed. Combining these ideas suggest that an appropriate data structure will be a binary tree reflecting the divide and conquer calls. At the bottom of the tree structure there is one leaf for each character in the pattern. The tree is formed by joining pairs of leaves and building up to a root which contains the score for the whole alignment. Each node of the tree will store enough information to compare the score for an alignment which includes the $l+1^{st}$ character of the text with the best score for alignments within the first l characters of the text. A fuller description of the data structure follows, but first we want to describe another metaphor for our heuristic.

Our metaphor is to consider the data structure as a walking tree. When the heuristic is considering position $l+1$ of the text, the leaves of the tree will be positioned over the $|P|$ contiguous characters up to and including character $l+1$. The leaves will also be remembering some of the information for the best alignment found so far. On the basis of this remembered information and the comparison with the character being looked at, the leaf will update its information and pass it to its parent. The data will percolate up to the root where the new best score is calculated. The tree can then walk to the next position by moving each of its leaves one character to the right. The whole text will be processed when the leftmost leaf has processed the rightmost character of the text.

Data Structure

The computations are carried out on a binary tree of depth $\lfloor \log(m) \rfloor$. Each node of the tree represents a substring of P. If a node represents the substring

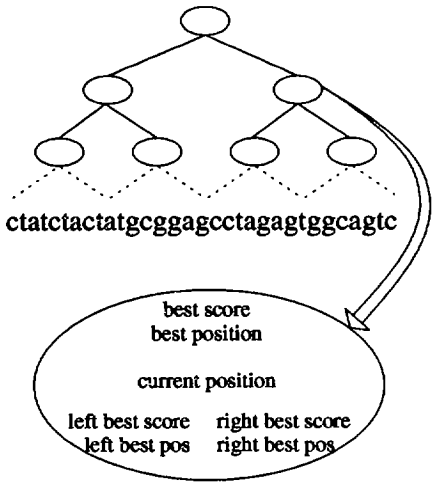

Figure 1: Data structure used to align the pattern within the text. In this picture, each leaf node represents 8 characters of the pattern, each of the internal nodes represents 16 characters of the pattern, and the root node represents the entire pattern. Each of the nodes contains the fields shown in the expanded node.

$p_i p_{i+1} \cdots p_{i+2^j-1}$, then the left child of the node represents the substring $p_i p_{i+1} \cdots p_{i+2^{j-1}-1}$ and the right child represents $p_{i+2^{j-1}} p_{i+1^{j-1}+1} \cdots p_{i+2^j-1}$. The root node of the tree represents the entire pattern string and the leaf nodes each represent a single character of the pattern. A graphical representation of the binary tree used to represent the pattern is given in Figure 1.

At each non leaf node we store the seven pieces of data described next. Best score (bs) is the best score found so far for aligning the substring represented by the node. Best position (bsp) is the position of the text with which the rightmost character of the pattern substring is aligned in the current best alignment. Current position (cp) is the position in the text which is being considered by the rightmost leaf associated with this pattern substring. Left best score (ls) is the best current score for aligning the left half of this pattern substring. Left best position (lsp) is the length of the left half of the pattern substring plus the text position with which the rightmost character of the left half of the pattern substring is currently aligned. Right best score (rs) is the best current score for aligning the right half of the pattern substring. Right best position (rsp) is the text position with which the rightmost character of the right half of the pattern substring is currently aligned.

Basic Heuristic

The score of each leaf node is computed as a function of the two characters p_i and t_{i+k}, $\mathcal{M}(p_i, t_{\chi_s(i)})$. The score of each of the non-leaf nodes is the maximum of the following three values.

1. $bs - \mathcal{G}(|cp - bsp|)$
2. $ls + rs - \mathcal{G}(|cp - rp|) - \mathcal{G}(|rp - lp|)$
3. $ls + rs - \mathcal{G}(|cp - lp|) - \mathcal{G}(|lp - rp|)$

The first value is the best score for the node minus the gap penalty for the difference between the current position and the best score position. The second value is the sum of the best left score and the best right score minus the gap between the current position and the right position minus the gap between the right position and the left position. The third value is the sum of the best left score and the best right score minus the gap between the current position and the left position minus the gap between the left position and the right position.

If the maximum of the three scores is larger than the best score minus the gap penalty between the best position and the current position, the best score is replaced with the maximum of the three scores and the best position is replaced with the current position.

These scores are computed level by level up the tree so that the root will eventually compute the best score for an alignment within the part of the text that has been considered. After the root has computed a score, each leaf is moved right by one text character, and the whole process of computing a score is repeated.

Including Inversions

To compute a score allowing inversions in the alignment we compute two alignment trees in parallel. The first tree is computed using the pattern and the second tree in computed using the inverse complement of the pattern. Each node in the first tree, representing a substring of the pattern, has a sister node in the second tree that represents the inverse complement of that substring. Each node has an eighth data value, inverse value (iv), that is the score of the node's sister node. Each node now computes its new best score by considering the four possible scores make up by using a right half substring from the node or its sister node coupled with a left half substring from the node or its sister node. The appropriate inversion penalty will be applied when a half substring from the sister node is used.

Constructing the alignment

The heuristics above compute an alignment score and position for a pattern in a text. To compute the alignment, we add a field, X, to each node. This field contains an alignment for the substring that the node represents. Each time the best score for the node is updated, the alignment of the left child is copied into the left half of X and the alignment of the right child is copied into the right half X. When the computation completes for the alignment of the pattern at position k of the text, the alignment associated with the root node is the alignment that produced the best score.

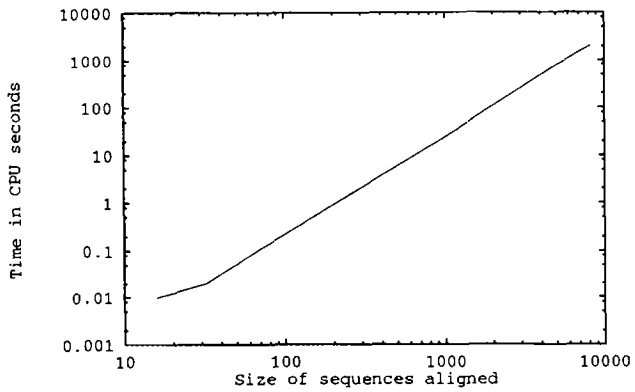

Figure 2: CPU time used by our heuristic to align pairs of equal length sequences on a Sun SPARC–10.

Resource Usage

We have shown the following resource usage results for the heuristic computing the score of an alignment with inversions in (Holloway 1992).

- The heuristic will execute in time proportional to the product of the length of the text and the length of the pattern.
- The work space used by the heuristic is proportional to the length of the pattern. The work space used is independent of the length of the text.
- The heuristic underestimates the actual alignment score of a pattern at a given position in the text by, at most, the sum of the gap penalties in the alignment at that position.

We have shown the following resource usage results for the heuristic constructing an alignment with inversions in (Holloway 1992).

- The heuristic to construct the alignment will run in $O(nm \log m)$ time when given T a text string of length n and P a pattern string of length m.
- The heuristic to construct the alignment will use $O(m \log m)$ work space when given T, a text string of length n and P, a pattern string of length m.

To time the heuristic, we constructed an alignment of equal sized regions of the variola virus genome. The size of the regions being aligned was varied from 16 bases to 8192 bases. Figure 2 shows the CPU time used by a Sun SPARC–10 to align the sequences. As expected from the results referenced above, the time used by the heuristic increases by a factor of four as the size of the sequences is doubled.

We have since implemented and optimized our heuristic on a single node of a Meiko CS–2 computer. A node consists of one Texas Instruments SuperSPARC scalar processor and two Fujitsu μVP vector processors. Using the optimized heuristic on one node of the Meiko CS–2, two 2^{13} base sequences can be aligned in less than 1.5 CPU minutes. Aligning a pair of sequences of length 2^{15} requires less than 25 minutes of CPU time. This agrees with the predicted run time which says that increasing both the pattern and the text by a factor of 4 should increase the run time by a factor of 16.

Picornavirus Alignments

Picornaviridae is a family of single stranded RNA viruses that are 7.2 to 8.4 kb in length. It is composed of the five genera Aphthovirus, Cardiovirus, Enterovirus, Hepatovirus, and Rhinovirus. The RNA typically codes for four major polypeptides and several proteases.

We selected the sequences from GenBank release 81.0 (February 1994) with the keywords "Picornaviridae", "complete", "genome", and "sequence". From these GenBank entries we selected the twenty entries that contained a complete Picornavirus genome sequence. Using the heuristic described above, we computed the alignment score between each pair of loci. We used the functions

$$\mathcal{M}(a,b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b \end{cases}, \qquad \begin{aligned} \mathcal{G}(n) &= -2 - \log n \\ \mathcal{I}(n) &= -4 - \log n \end{aligned}$$

to evaluate alignments. The alignment score, s, is converted to a distance, d, using

$$d = 1 - \frac{s}{\max_s}$$

where $\max_s$ is the maximum possible alignment score for the pattern sequence. We then used the Fitch–Margoliash distance matrix method as implemented by Joe Felsenstein in the Phylip 3.53c package to construct the phylogenetic tree in Figure 3.

The phylogeny presented in Figure 3 is based on the complete viral genomes and is nearly identical to the phylogeny presented by (Stanway 1990) based on the P1 (capsid–encoding) regions of each genome. The Hepatovirus genera (HPAACG, HPACG, HPA, HPAA) cluster tightly as expected since they are nearly identical sequences. The Cardioviruses (EVCGAA, EMCDCG, EMCBCG, MNGPOLY, TMECG, TMEPP, TMEVCPLT, and TMEGDVCG) form three groups, the encephalomyocarditis viruses, a mengovirus, and the Theiler murine encephalomyelitis viruses. The Enteroviruses (SVDG, CXB5CGA, CXB4S, CXA21CG, POLIOS1, CXA24CG, and BEVVG527) cluster loosely. The Rhinovirus (HRV) is not near any of the other Picornaviruses.

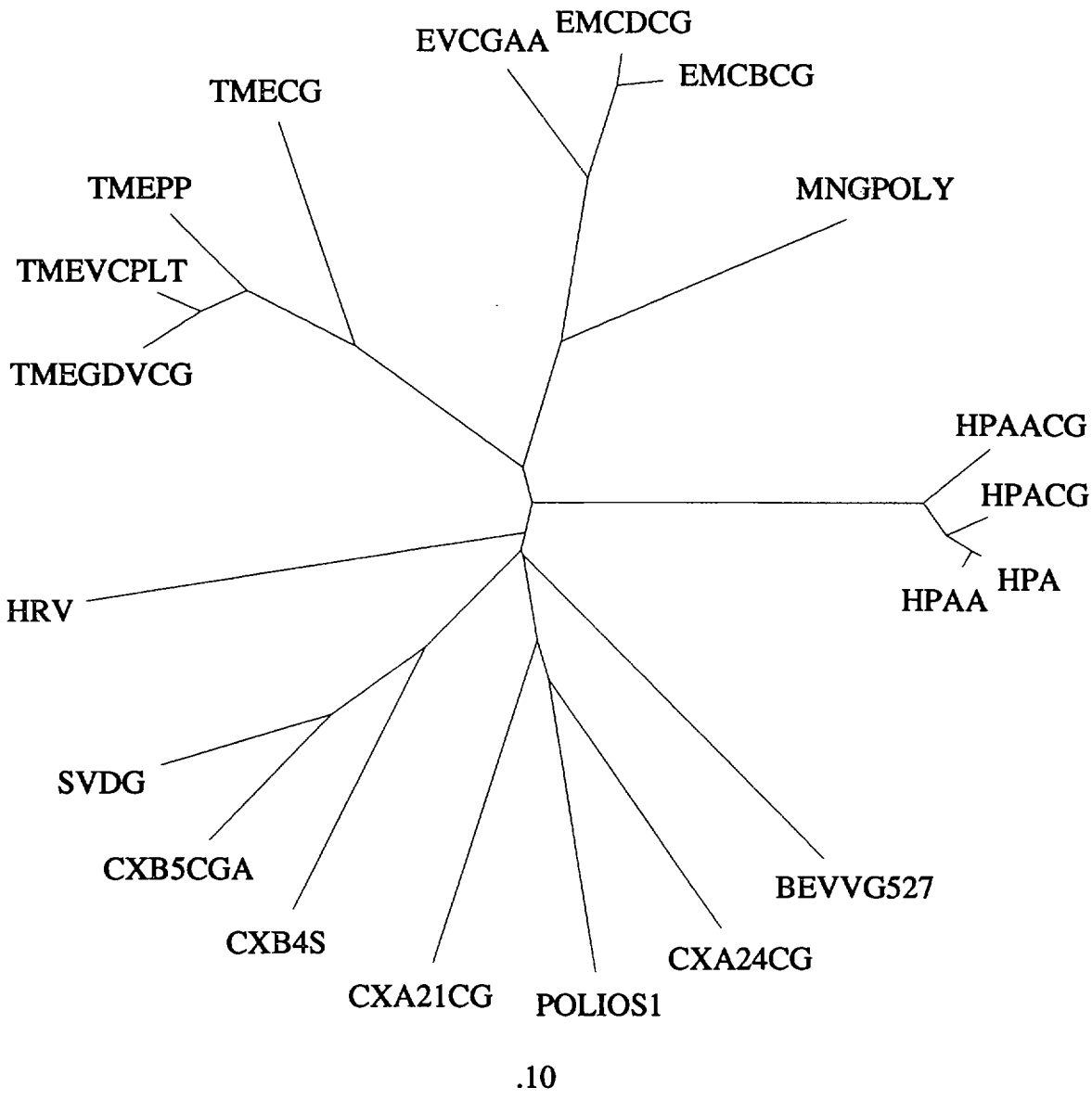

Figure 3: Phylogenetic tree of the Picornavirus constructed using distances between the complete genome sequences as computed by our heuristic.

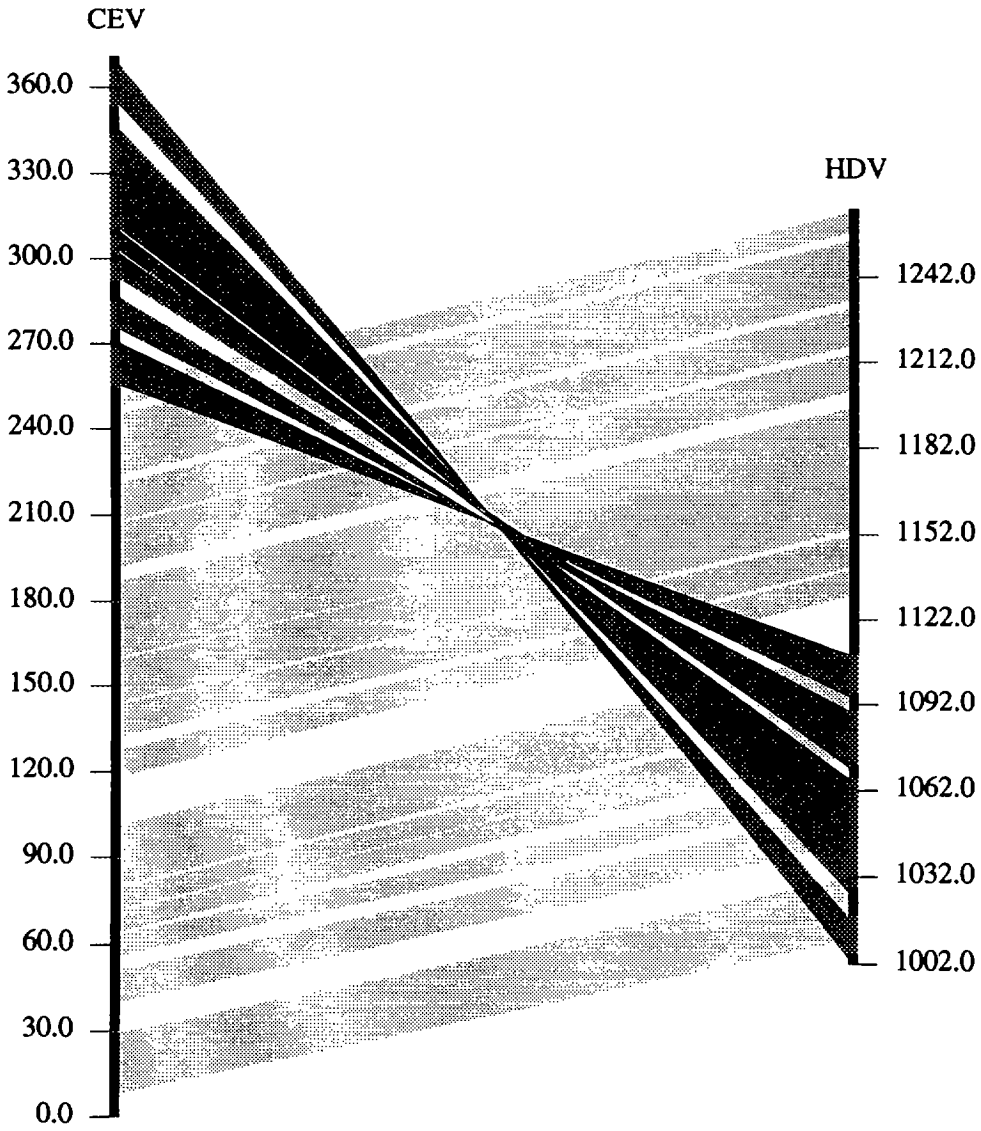

Figure 4: Alignment of the citrus exocortis viroid genome and the hepatitis delta virus genome. Light gray indicates regions of direct alignment and black indicates regions where the inverted complements align.

We used our heuristic to align the hepatitis delta virus (HDV) genome (GenBank accession D01075) with the citrus exocortis viroid (CEV) genome (GenBank accession K00965) to construct the alignment in Figure 4. The complement of the bases 1013...1600 of the HDV genome code for the hepatitis delta–antigen. Our heuristic aligns CEV with the hepatitis delta–antigen region of HDV. Note that the inverse complement of the 3′ end of the CEV matches the same region as the 5′ end of CEV.

Applications

There are at least two areas in the biological sciences that our heuristic will directly impact. First, searching for and aligning two or more genetic sequences (DNA, RNA, Protein) where parts of one sequence may be rearranged with swaps and inverse complements with respect to the other sequence. Second, our heuristic can be used to discover chromosomal rearrangements by examining genetic maps. Our heuristic can also be applied to the problems of constructing evolutionary trees, discovery of common motifs, and multidimensional pattern matching.

In their recent paper, Devos et al. (Devos *et al.* 1993), present evidence for multiple evolutionary translocations that differentiate the rye and wheat genomes. Using restriction fragment length polymorphism (RFLP) and other methods, they found markers in both the rye and wheat genomes and used them to construct a series of translocation events for the evolution of the rye genome from the wheat genome. Given the sequence of markers on the chromosomes of wheat and rye, our heuristic could automatically construct an alignment of one genome with the other indicating the regions of the genome where translocation events have occurred.

Our heuristic can be expanded from one dimensional approximate matching to multi dimensional approximate pattern matching. For two or three dimensional approximate pattern matching such as image analysis, operators such as rotate, expand, and shrink can be used in addition to the swap and invert operators. Three dimensional approximate matching could be used to search the 3D Protein structure database (PDB) for similar structures.

Phylogenetic trees are often constructed using an alignment of a single homologous region of DNA or RNA from each of the organisms being compared. A different approach is taken by Sankoff et. al. (Sankoff *et al.* 1992) who use the known gene order of entire organellar genomes to construct a phylogenetic tree. Our heuristic can be applied to constructing phylogenetic trees in two ways. First, when the order of the genes (or other conserved regions) is not known for each genome, our heuristic will be able to construct the order. Secondly, when the number of genes is too large for the methods described by Sankoff *et. al.*, our heuristic will be able to find the approximate minimum number of swaps and inversions required to transform one genome into the other.

Conclusion and Future Research

There is evidence that evolution may proceed by moving and inverting segments of a genome in addition to changing, inserting, and deleting individual bases in the genome. The heuristics that are currently in use to align genetic sequences frequently fail to consider the swap and invert operations. We have introduced a heuristic to allow the comparison of gene sequences using both types of operations; the change, insert, and delete operations on individual bases, and the move and invert operations on segments of the sequences.

We have described a simple heuristic to find alignments between strings allowing swaps and inversions. The heuristic runs quickly, in time proportional to the product of the lengths of the sequences being aligned and uses a very small amount of work space, proportional to the length of the shorter sequence.

It may be possible to construct a "smart" disk controller based on our divide and conquer heuristic. The heuristic uses only a few simple operations and never needs to back up in the text. Each processor would need a small, constant sized memory, and would need to communicate with at most four other processors. Such a disk controller would allow database search heuristics to start with only the sequences that are similar to the query sequence.

The divide and conquer approach should be easy to apply to multi–dimensional approximate matching. The idea of inverted substrings in the one dimensional case can be extended to transformations of sub–patterns in the multidimensional case.

An implementation of the heuristics that have been described in this paper is available via anonymous ftp in the file `holloway/dnc.tar.Z` from the `bcc.orst.edu` computer.

Acknowledgments

This work was supported in part by NSF grant CDA–9216172.

References

Bertossi, A. A.; Lodi, E.; Luccio, F.; and Pagli, L. 1990. Context–dependent string matching. In Capocelli, R. M., ed., *Sequences, combinatorics, compression, security and transmission*, 25–40. Springer–Verlag.

Devos, K. M.; Atkinson, M. D.; Chinoy, C. N.; Francis, H. A.; Harcourt, R. L.; Koebner, R. M. D.; Liu, C. J.; Masojc, P.; Xie, D. X.; and Gale, M. D. 1993. Chromosomal rearrangements in the rye genome relative to that of wheat. *Theoretical and Applied Genetics* 85:673–680.

Eppstein, D.; Galil, Z.; and Giancarlo, R. 1990. Efficient algorithms with applications to molecular biology. In Capocelli, R. M., ed., *Sequences, combinatorics, compression, security and transmission*, 59–74. Springer–Verlag.

Galil, Z., and Giancarlo, R. 1989. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science* 65:107–118.

Goad, W. B. 1986. Computational analysis of genetic sequences. *Annual Review of biophysics and biophysical chemistry* 15:79–95.

Holloway, J. L. 1992. *String Matching Algorithms with Applications in Molecular Biology.* Ph.D. Dissertation, Oregon State Universtiy, Corvallis, OR.

Lowrance, R., and Wagner, R. A. 1975. An extension of the string to string correction problem. *Journal of the Association for Computing Machinery* 23(2):177–183.

Miller, W., and Myers, E. W. 1988. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology* 50:97–120.

Sankoff, D., and Goldstein, M. 1989. Probabilistic models of genome shuffling. *Bulletin of Mathematical Biology* 51:117–124.

Sankoff, D.; Leduc, G.; Antoine, N.; Paquin, B.; Lang, B. F.; and Cedergren, R. 1992. Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome. *Proceedings of the National Academy of Science* 89:6875–6579.

Schoniger, M., and Waterman, M. S. 1992. A local algorithm for DNA sequence alignment with inversions. *Bulletin of Mathematical Biology* 54(4):521–536.

Stanway, G. 1990. Structure, function, and evolution of picornaviruses. *Journal of General Virology* 87:2483–2501.

Tichy, W. F. 1984. The string to string correction problem with block moves. *ACM Transactions on Computer Systems* 2(4):309–321.

Waterman, M. S. 1984. General methods of sequence comparison. *Bulletin of Mathematical Biology* 46:473–500.

Waterman, M. S. 1988. Computer analysis of nucleic acid sequences. *Methods in Enzymology* 164:765–795.