JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 18, Number 5, 2011
© Mary Ann Liebert, Inc.
Pp. 705–728
DOI: 10.1089/cmb.2010.0101

A Theoretical Model for Whole Genome Alignment

NAHLA A. BELAL[1] and LENWOOD S. HEATH[2]

ABSTRACT

We present a graph-based model for representing two aligned genomic sequences. An alignment graph is a mixed graph consisting of two sets of vertices, each representing one of the input sequences, and three sets of edges. These edges allow the model to represent a number of evolutionary events. This model is used to perform sequence alignment at the level of nucleotides. We define a scoring function for alignment graphs. We show that minimizing the score is NP-complete. However, we present a dynamic programming algorithm that solves the minimization problem optimally for a certain class of alignments, called breakable arrangements. Algorithms for analyzing breakable arrangements are presented. We also present a greedy algorithm that is capable of representing reversals. We present a dynamic programming algorithm that optimally aligns two genomic sequences, when one of the input sequences is a breakable arrangement of the other. Comparing what we define as breakable arrangements to alignments generated by other algorithms, it is seen that many already aligned genomes fall into the category of being breakable. Moreover, the greedy algorithm is shown to represent reversals, besides rearrangements, mutations, and other evolutionary events.

Key words: algorithms, combinatorics, probability, sequences.

1. INTRODUCTION

Synteny is a slippery concept. In some contexts, it is used to denote the collocation of several genetic loci on one chromosome. In other contexts, it is used to describe the preserved order of genes of related species on chromosomes. Most research simplifies analysis of a genome to the level of "gene" or "synteny block." This simplification has a number of problems. It is difficult to know whether a sequence is actually a gene and, if it is a gene, where the start codon, exons, introns, and stop codon are. In addition, a gene may have multiple transcripts, a sequence may be a pseudogene, and the sequences of two genes may overlap. Especially ignored is the absolute location of genes on a chromosome, as opposed to the less informative relative position.

A new synteny-like term could be defined to compare genomes at the detailed level of nucleotides, taking into account the length of genes and their absolute positions. One could think of genomes as being composed of parts (subsequences) and compare genomes by comparing the parts that constitute each genome. However, the parts are not defined in advance. Therefore, an algorithm has to define the parts so as

[1]Department of Computer Science, AAST, Alexandria, Egypt.
[2]Department of Computer Science, Virginia Tech, Blacksburg, Virginia.

to optimize a similarity measure. Given two genomes with parts that are very similar and parts that are not, an algorithm may seek to infer how similar the two genomes are. This takes us to our concept of whole genome alignment.

Let G_1 and G_2 be two genomes. We wish to define a distance between G_1 and G_2 based on an optimal alignment. Here, alignment is something more than the traditional pairwise or multiple sequence alignment. Suppose G_1 and G_2 are identical, except for a substring in the middle of G_2 that is the Watson-Crick complement of the corresponding substring in G_1. In genome rearrangement terms, this is a reversal. Now, real genomes do not evolve so neatly. There will not be crisp boundaries for the reversal. And, there will be point mutations that occur, as well as sequence insertions and deletions. However, ultimately, these two genomes have a common ancestor genome and the parts of the current genome could, in principle, be ''mapped'' or ''aligned'' to parts of the ancestor and hence the two genomes could be piecewise aligned to each other.

Because of genome rearrangements and the large sizes of most genomes, an optimal alignment of the suggested sort will be expensive to compute. However, practical approximation algorithms can be developed. A whole genome alignment looks for a decomposition of G_1 and a decomposition of G_2 into substrings such that paired substrings are optimally aligned. It is possible that, in alignment, some strings will undergo Watson-Crick complementation at the nucleotide level. Two other scenarios can occur. One scenario is that there may be gene duplications that separate the two genomes. In this case, multiple sequences from one genome will align to a single sequence in the other genome. The other scenario is actual movement within the genome. A gene sequence or other genomic sequence may move from one part of a genome to another. The sequence may also be reversed (on the opposite strand) from where it started. In these cases, there are ''breaks'' forced in the alignment.

In this article, an alignment graph, a graph-based model, is presented and used for aligning two sequences. Moreover, the manner in which different evolutionary events are represented using the described model is presented. Also, a class of sequences—breakable arrangements—is defined for which the given dynamic programming algorithm gives optimal results, and a greedy heuristic is also given.

In Section 2, we present related work from the literature. Section 3 gives the preliminaries and definitions used in the rest of the article. Then, in Section 4, we show how the defined alignment graph is used to represent different evolutionary events, and how the graph is scored is discussed in Section 5. The problem addressed in this article is defined and proven to be NP-complete in Sections 6 and 7, respectively. A dynamic programming algorithm for solving the defined problem is presented in Section 8. Breakable arrangements are discussed in Section 9. Sections 10 and 11 give algorithms for identifying and counting breakable arrangements. Then, the dynamic programming algorithm is proven optimal for breakable arrangements in Section 12. In Section 13, real alignments are tested for breakability. A heuristic greedy algorithm is presented in Section 14 to solve the defined alignment problem. Finally, conclusions and future directions are presented in Section 15.

2. RELATED WORK

Sequence alignment is a way of arranging genomic sequences to identify similarities between subregions that point to some functional, structural, or evolutionary relationship. If two sequences in an alignment share a common ancestor, then differences between the two sequences could be interpreted as point mutations, insertions, deletions, or other evolutionary events that help us infer the evolutionary distance between the two sequences. Pairwise alignment techniques align two input sequences, whereas multiple sequence alignment techniques support three or more sequences.

According to Blanchette (2007), two strategies are known to solve alignment problems, namely, global alignment and local alignment. In global alignment, the sequences are considered as a whole. This imposes the constraint on the alignment that orthologous genes must be colinear, which prevents the detection of rearrangements and duplications. A general global alignment technique is the Needleman-Wunsch dynamic programming algorithm (Needleman and Wunsch, 1970). Local alignment considers fragments of the input sequences and aligns fragments rather than whole sequences, which overcomes the deficiency of global alignment that colinearity must be maintained, at the expense of efficiency, where a higher probability of false alignments is expected. The Smith-Waterman dynamic programming algorithm (Smith and Waterman, 1981) is a general local alignment algorithm. Some techniques combine both global and local alignment strategies, such as the hybrid technique named glocal (Brundo et al., 2003). For large sequences, exact methods are impractical, requiring large amounts of memory and long execution times. Many heuristics have been developed to overcome the impracticality of exact algorithms.

Some pairwise alignment methods are highlighted. Dot matrix methods construct plots with dots representing matching characters, where one sequence is placed at the topmost row, and the other sequence is placed on the leftmost column. This method is time consuming, but it is simple and easy to visualize. Some tools were developed using this technique, including the DNADot[1] web-based tool and the DOTLET[2] Java based tool (Junier and Pagni, 2000). These are both global alignment techniques. Another technique is dynamic programming, which can be used for both global and local alignments, as previously mentioned. Dynamic programming techniques use scoring functions to find optimal solutions, and once a scoring function is defined, a dynamic programming algorithm is guaranteed to find an optimal answer, if the scoring function defined is summed column-wise.

There are several techniques for multiple sequence alignment. Some of the techniques are sequence-based, like CLUSTAL W (Thompson et al., 1994), whereas others use secondary structures, like MUMMALS (Pei and Grishin, 2006), or 3D structures, as in M-Coffee (Wallace et al., 2006). Some are genome aligners, for example, MUMmer (Delcher et al., 1999). There are programs that use seeded pairwise alignment; these programs use heuristics for aligning large sequences, where a seed is defined as a short highly conserved match, and a local alignment is considered only if it contains this seed. Nearly all seeded pairwise alignment programs are based on BLAST (Altschul et al., 1990). Examples of programs that employ this technique are BLASTZ (Schwartz et al., 2003), LAGAN (Brundo et al., 2003), CHAOS (Brudno et al., 2003), AVID (Bray et al., 2003), and MUMmer (Delcher et al., 1999). Other programs that perform multiple sequence alignment either use multiple pairwise alignments or perform alignment on all input sequences at once. MLAGAN (Brundo et al., 2003), CLUSTAL W (Thompson et al., 1994), and MAVID (Bray and Pachter, 2004) use progressive sequence alignment (Durbin et al., 1998), where a phylogenetic tree is first inferred by performing pairwise alignments. A non-phylogenetic alignment technique is consistency-based multiple sequence alignment (Morgenstern et al., 2005, Pohler et al., 2005, Szklarczyk and Heringa, 2006, Ye and Huang, 2005). Also, the MAUVE tool (Darling et al., 2004) allows alignments with rearrangements. However, the results obtained by MAUVE were shown to be best on closely-related organisms. It also does not support the alignment of large regions shared by subsets of the genomes or the rearranged regions shared by subsets of the genomes. Moreover, it does not perform well when there are many duplicated segments. Another drawback of the MAUVE algorithm is that it requires manual data entry for some parameters. The progressive version of the MAUVE algorithm allows for alignment of more divergent genomes, and it reduces the manual adjustment of the alignment scoring parameters. It also aligns regions conserved among subsets of the input genomes. However, it still has the limitation of being slow, consuming more memory than the original MAUVE algorithm, and continuing the use of manual adjustment.

Other research directions define the alignment problem in terms of a graph problem. The local multiple sequence alignment problem could be viewed as finding Eulerian paths in a graph (Zhang and Waterman, 2005). Raphael et al. (2004) use de Bruijn graphs to perform multiple sequence alignments. They present a technique, A-Bruijn Alignment, which represents an alignment as a directed graph, and they present methods to detect cycles and reversals. The first task is to find a graph that represents the domain structure, and the second is to find a mapping of each sequence to this graph. The graph is constructed from a set of pairwise local alignments. The graph representation presented in Raphael et al. (2004) is used in the alignment of protein sequences with shuffled or repeated domain structure and also in the alignment of proteins containing domains that are not present in all proteins, domains that are present in different orders in different proteins, and domains that are present in multiple copies in some proteins. Moreover, the technique they present detects duplications and inversions.

Phuong et al. (2006) present an algorithm, ProDA, for aligning protein sequences with repeated and shuffled domains. The algorithm they present computes local alignments for every pair of sequences, then clusters those alignments into blocks of globally alignable subsequences to determine block boundaries and resolve inconsistencies between pairwise alignments to be able to find the multiple alignment between blocks.

[1]Available at http://www.vivo.colostate.edu/molkit/dnadot/.
[2]Available at http://myhits.isb-sib.ch/cgi-bin/dotlet.

Paten et al. (2008) present two programs, namely Enredo and Pecan, for multiple genome alignment. They divide the problem of multiple genome alignment into two stages. The first stage partitions the input genomes into a set of colinear segments; this is carried out by the program Enredo, which handles rearrangements and duplications. The second stage generates a base pair level alignment map for each colinear segment; this is carried out by the program Pecan, which makes the alignment problem practical on a large scale.

Ma et al. (2008) present a polynomial-time algorithm to find the most parsimonious evolutionary history of any set of related genomes. They start with a single genome, called the root genome, taken from a species called the original species. Evolution of the root genome takes place through evolutionary events, namely, loss and gain of chromosomes, duplication, and rearrangement. When a speciation event occurs, an identical copy of the genome is made, and then evolution of the copies takes place independently.

Ergun et al. (2003) present a linear time greedy algorithm that computes sequence similarity with rearrangements. They define two edit operations, character edits, which allow insertions, deletions and replacements, and segment edits, which allow substring relocations, deletions, and duplications. The distance between two sequences is the minimum number of edit operations needed to transform one string into the other.

An algorithm for multiple genome alignment without a reference organism is implemented as part of the VISTA genome pipeline (Dubchak et al., 2009). The algorithm is based on progressive alignment. After aligning two genomes, the algorithm builds synteny blocks based on the outgroups, where outgroups are the genomes that are not yet aligned. This helps the algorithm then align more distant genomes.

In Yancopoulos et al. (2005), a method for finding genomic distances is presented. The method is based on a comparison graph generated for two genomes, and the distance is calculated from breakpoints and cycles in the graph. In their method, they also define a double-cut-and-join operation that accounts for the events of inversion, translocation, fission, and fusion.

Otu and Sayood (2003) propose a new sequence distance measure. Their method uses Lempel-Ziv complexity for finding the relative distances between sequences, and they use the distance matrix obtained to construct phylogenetic trees.

Varre et al. (1999) present another family of genome distances, namely transformation distances. Transformation distances are calculated in terms of segment-based events, like insertion and deletion of sequence segments. The algorithm presented computes the exact distance between two input sequences, without taking the order of residues into account, and, hence, the algorithm is able to account for duplications and translocations.

3. PRELIMINARIES

Let $S_1 = a_1a_2 \cdots a_m$ and $S_2 = b_1b_2 \cdots b_n$ be two genomic sequences. We define a class of edge-colored, mixed directed and undirected graphs representing alignments of S_1 and S_2. There are two disjoint sets of nodes, $U = \{u_1, u_2, \ldots, u_m\}$ and $V = \{v_1, v_2, \ldots v_n\}$. Node u_i is labeled a_i, and node v_i is labeled b_i. The directed edges are colored blue and form the following set of edges:

$$E_{\text{blue}} = \begin{array}{c} \{(u_i, u_{i+1}) | 1 \leq i < m\} \cup \\ \{(v_i, v_{i+1}) | 1 \leq i < n\}. \end{array}$$

There are two sets of undirected edges. E_{black} consists of edges colored black, each of which connects a node in U to a node in V. E_{red} consists of edges colored red, each of which connects a node in U to a node in V. For each choice of E_{black} and E_{red}, the resulting mixed graph $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$ is an *alignment graph* for S_1 and S_2. The blue edges form two directed paths, one for each genomic sequence. Intuitively, a black edge connects two nucleotides that are on the same strand, while a red edge connects two nucleotides that are on opposite strands. In other words, black edges connect nucleotides that are not complementary, while red edges connect complementary nucleotides. A node that has no black or red edge incident on it is *unaligned.* Figure 1 shows an example of an alignment graph.

Let $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$ be an alignment graph for S_1 and S_2. A *free node* is one that is unaligned, that is, having neither black nor red incident edges. Two edges e_{ij} and e_{kl} in G are *adjacent in* S_1 when nodes i and k in S_1 are separated by zero or more free nodes, namely $u_{i+1}, u_{i+2}, \ldots, u_{k-1}$, and nodes

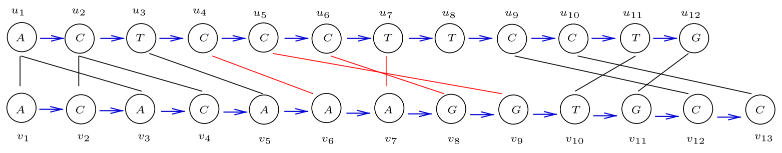
u_1 u_2 u_3 u_4 u_5 u_6 u_7 u_8 u_9 u_{10} u_{11} u_{12}
A C T C C C T T C C T G
A C A C A A A G G T G C C
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13}

FIG. 1. The effect of evolutionary events on an alignment graph.

j and l in S_2 are also separated by zero or more free nodes, namely $v_{j+1}, v_{j+2}, \ldots, v_{l-1}$. Similarly, two edges e_{ij} and e_{kl} in G are *adjacent in* S_2 when nodes j and l in S_2 are separated by zero or more free nodes, namely $v_{j+1}, v_{j+2}, \ldots, v_{l-1}$, and nodes i and k in S_1 are also separated by zero or more free nodes, namely $u_{i+1}, u_{i+2}, \ldots, u_{k-1}$. In case of red edges, free nodes in S_1 are $u_{i-1}, u_{i-2}, \ldots, u_{k+1}$ and in S_2 are $v_{j-1}, v_{j-2}, \ldots v_{l+1}$. A *break* in the alignment graph G occurs when there exist two adjacent edges in S_1 or in S_2, where one edge is in E_{black} and the other is in E_{red} or vice versa. A *break* also occurs when two edges are adjacent in S_1 but not in S_2 or vice versa.

A *point mutation* is represented by either a black edge connecting two nodes with different labels, or a red edge connecting two nodes with labels that are not complementary.

A *duplication* is a node with more than one incident edges, either red or black or both.

4. REPRESENTING DIFFERENT PHENOMENA

Our model aligns at the nucleotide level. This makes it possible to align two sequences without the need to identify a gene sequence, start/stop codons, exons, introns, and pseudogenes. The model presented is a graph-based model that uses three kinds of edges, blue, black, and red edges, contained in E_{blue}, E_{black}, and E_{red}, respectively. Blue edges sequence the nucleotides. Black and red edges make it possible to represent evolutionary events other than the insertions, deletions, and mutations identified by other alignment techniques. For example, the model allows the representation of rearrangements, reversals, and duplications.

Consider the graph in Figure 1. The nodes in the top row represent the first sequence, while those in the bottom row represent the second sequence. The blue edges between nodes of the same sequence indicate the order of nucleotides in the sequence. A black edge connecting a node from the top row to a node from the bottom row represents a position in both input sequences that descends from the same position in the least common ancestor and that are on the same strand. Analogously, a red edge represents a position in both input sequences that descends from the same position in the least common ancestor and that are on opposite strands. Consider the effect of different evolutionary phenomena on the graph in Figure 1. First, the events of insertion and deletion result in repositioning of the nucleotides and having nodes that are unaligned. Therefore, the effect of insertion and deletion is that we add or remove blue directed edges. In the case of insertion, undirected edges are not affected, while deletion may result in deleting undirected edges as well. In Figure 1, the node u_8 can be the result of an insertion in the first sequence, or a deletion from the second sequence.

In case of duplication, the duplicate nodes are aligned to the same nodes in the other sequence; this gives a one-to-many relation, as shown in Figure 1 for nodes u_1 and u_2, which are repeated twice in the second sequence, in the nodes v_1, v_2, v_3, and v_4. New blue edges are added, as well as black edges to align the duplicate nodes.

Point mutations do not affect the edges of the graph. Since black and red edges indicate common ancestry, mutation does not remove this common ancestor. This is illustrated in Figure 1, once for nodes u_3 and v_5, and another time for nodes u_6 and v_7, where the red edge indicates that these nodes should be complementary.

In reversals, we have to represent complementation, that is, the movement of a subsequence to the opposite strand. We have a new order for the nucleotides, along with complementation of nucleotides on the opposite strand. The effect on the graph is only in the position of edges, but they still connect the same nucleotides they connected before the reversal. This is shown in Figure 1, in the nodes u_5, u_6, u_7 and v_7, v_8, v_9.

Transposition results in repositioning a subsequence, and, hence, the edges are repositioned accordingly, as shown in Figure 1 for nodes u_9, u_{10}, u_{11}, u_{12} and v_{10}, v_{11}, v_{12}, v_{13}. The blue edges indicate the change in location, and, hence, the black edges are also repositioned.

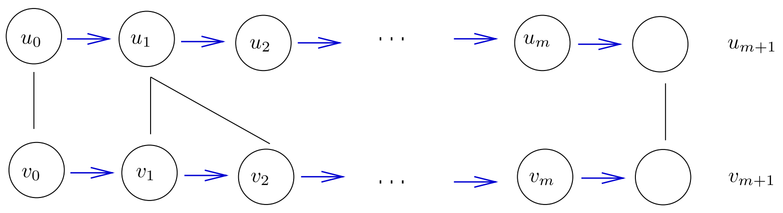

FIG. 2. An alignment graph that starts by a duplication.

5. SCORING MECHANISM

Given an alignment graph $G = (U, V; E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$, we must calculate a score $s(G)$ that represents the level of alignment intrinsic in G. The score $s(G)$ can depend on the following components of G:

- The presence, absence, and color of edges.
- The indices of the nodes.
- The labels of the connected nodes (u_i and v_j).
- The number of edges incident to each node.

Special cases can arise at the ends of a sequence, for example, if a sequence starts by a duplication. Therefore, for the purposes of scoring, two additional nodes are added to each sequence, one at the beginning and another at the end. Hence, given two sequences $U = u_1, \ldots, u_m$ and $V = v_1, \ldots, v_n$, nodes u_0 and u_{m+1} are added to U, and nodes v_0 and v_{n+1} are added to V, where an alignment graph connects node u_0 to v_0 and node u_{m+1} to v_{n+1} using black edges (Fig. 2).

Those graph characteristics guide us to calculate an alignment score that accounts for different evolutionary events. Those events result in the graph characteristics defined previously, namely, breaks, mutations, free nodes, and duplications.

Given two identical sequences, an optimal alignment is a *perfect alignment,* that has $E_{\text{black}} = \{(u_i, v_i) \mid 1 \leq i \leq m\}$ and $E_{\text{red}} = \emptyset$. A perfect alignment has a score $s(G) = 0$ and only a perfect alignment should have $s(G) = 0$. Each break, mutation, free node, and duplication adds a penalty to $s(G)$. For each break, we add a penalty w_b to the score; for mutations, we add w_m; for a free node, we add w_f; and for duplications, we add w_d. This yields the following formula

$$s(G) = bw_b + mw_m + fw_f + dw_d,$$

where b is the number of breaks, m is the number of mutations, f is the number of free nodes, and d is the number of duplications.

Figure 3 is an example to illustrate the described scoring mechanism. Let $w_b = 1$, $w_m = 4$, $w_f = 4$, and $w_d = 4$. $U = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7\}$ and $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$. Three breaks can be detected, at (u_1, v_1), (u_4, v_2), and (u_5, v_5). Also, a mutation is seen between nodes u_6 and v_7. Finally, there is one free node, v_6. This results in $s(G) = 3w_b + w_m + w_f = 11$.

6. PROBLEM DEFINITION

Given a scoring function $s(G)$ for alignment graphs, we have the following computational problem.

Optimal Whole Genome Alignment Graph
INSTANCE: Two DNA sequences S_1 and S_2 and weights w_b, w_m, w_f, and w_d.
SOLUTION: An alignment graph G for S_1 and S_2 that minimizes the alignment score, $s(G)$.

This is the computational problem that we address.

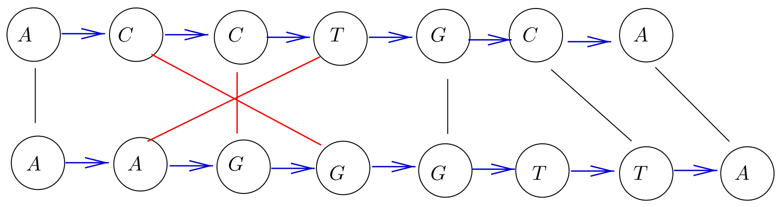

FIG. 3. Scoring example.

7. NP-COMPLETENESS PROOF

The following 3-Partition problem is used in the proof that Optimal Whole Genome Alignment Graph is NP-complete (Garey and Johnson, 1979).

3-Partition
INSTANCE: Set $\mathbb{A}$ of $3m$ elements, a bound $B > 0$, and a positive size $s(a)$ for each $a \in \mathbb{A}$ such that $B/4 < s(a) < B/2$ and such that $\sum_{a \in \mathbb{A}} s(a) = mB$.
QUESTION: Can $\mathbb{A}$ be partitioned into m disjoint sets $\mathbb{A}_1, \mathbb{A}_2, \ldots, \mathbb{A}_m$, each containing 3 elements, such that, for $1 \leq i \leq m$, we have $\sum_{a \in \mathbb{A}_i} = B$?

The 3-Partition problem is NP-complete in the strong sense, which means that it remains NP-complete if we express the numbers in the instance in unary.

Theorem 1. *Optimal Whole Genome Alignment Graph is NP-complete.*

Proof. We first need to show that Optimal Whole Genome Alignment Graph is in NP. Given a whole genome alignment, it is straightforward to compute, in polynomial time, the score of the alignment and check whether that score is less than the given bound. Hence, Optimal Whole Genome Alignment Graph is in NP.

To complete the proof, we reduce 3-Partition to Optimal Whole Genome Alignment Graph in polynomial time. Let $\mathbb{A} = \{a_1, a_2, \ldots, a_{3m}\}$, a set of $3m$ elements, sizes $s(a_i)$ for each $a_i \in \mathbb{A}$, and bound B constitute an instance of 3-Partition. We assume that all the numbers are represented in unary. The corresponding instance of Optimal Whole Genome Alignment Graph consists of two sequences over the DNA alphabet:

$$U = A^{s(a_1)} T A^{s(a_2)} T \cdots A^{s(a_{3m})} G^{m-1}$$
$$V = (A^B G)^{m-1} A^B T^{3m-1}.$$

Both strings have length $mB + 4m - 2$ and the same counts of A's, G's, and T's. Set the weights for scoring an alignment to $w_b = 1$ and $w_m = w_f = w_d = 4$. The bound for the score is $S = 7m - 3$. It is clear that U, V, and S can be constructed from the instance of 3-Partition in polynomial time, since the sizes are represented in unary.

If only black edges are allowed, then there are $7m - 3$ "natural" breaks in U, consisting of one before and after each of $3m - 1$ T's and one before each of $m - 1$ G's. If there is a 3-partition of $\mathbb{A}$, then it is straightforward to give a global alignment of U and V of score S: use the partition of $\mathbb{A}$ to match three blocks of A's in U to each A^B block in V, then match each G in U to an arbitrary G in V and each T in U to an arbitrary T in V.

Now assume that there is a global alignment of U and V with score $\leq S$. It is clear that, unless there are red edges, then the global alignment has at least S breaks (all of the natural ones in U). Since we assume that there are no red edges, there can be no mutations, free nodes, or duplicate nodes. Hence, each block of A^B in V specifies three elements of $\mathbb{A}$, and we get the desired 3-partition of $\mathbb{A}$.

If only red edges are allowed, we have the same number of natural breaks, using the same construction above, since there is one break before and after each delimiter, either a T or a G, then the score will still be $7m - 3$. When both black and red edges are allowed, then some of the T's will connect to A's using red edges. Therefore, there will be the same two breaks at each delimiter T connected to an A. In addition to that, there will be a break at each of the A's that will be connected to one of the T's at the end of the second sequence. Therefore, if the number of T's connected to A's by red edges is k, then the number of breaks will be $7m - 3 + 2k$. This shows that allowing red edges will never result in a score less than $7m - 3$.

We have demonstrated that 3-Partition reduces to Optimal Whole Genome Alignment Graph in polynomial time. We conclude that Optimal Whole Genome Alignment Graph is NP-complete. ■

8. DYNAMIC PROGRAMMING

A dynamic programming algorithm can be used to approximately solve reversal-free and duplication-free whole genome alignment. The presented dynamic programming algorithm is shown optimal for a class of sequences defined later in Section 9. The algorithm takes two sequences S_1 and S_2 as input, of lengths m

FIG. 4. Algorithm for RegAlign.

```
1 RegAlign($X[1,m]$, $Y[1,n]$)
2 for $i \leftarrow 0$ to $m$
3     do $OPT(i,0) \leftarrow \sum_{k=1}^{i} s(x_k,-)$
4 for $j \leftarrow 0$ to $n$
5     do $OPT(0,j) \leftarrow \sum_{k=1}^{j} s(-,y_k)$
6 for $i \leftarrow 0$ to $m$
7     do for $j \leftarrow 0$ to $n$
8         do $OPT[i,j] \leftarrow$
              $\min \{s(x_i,-) + OPT[i-1,j],$
                  $s(-,y_j) + OPT[i,j-1],$
                  $s(x_i,y_j) + OPT[i-1,j-1]\}$
9 return $OPT(m,n)$
```

and n, respectively. The algorithm starts processing all the pairs of subsequences, starting from pairs of length 0 and increasing the lengths until m and n. Finding the alignment score between pairs of length 1 is trivial, and the score can be calculated as follows. $S_1[i, i]$ and $S_2[j, j]$ are either identical with a score of 0, or different and considered a mutation with a score of w_m.

The alignment scores of the pairs of substrings are stored in an array, AlignScore, where each cell in the array is indexed by four indices, i, k, j, and l, where i indicates the S_1 subsequence starting position, k indicates the S_1 subsequence ending position, j indicates the S_2 subsequence starting position, and l indicates the S_2 subsequence ending position. Processing pairs of subsequences continues to longer subsequences, where the score is calculated using the previously calculated scores of shorter subsequences. This is done by trying each and every possible break, and aligning all possible parts together, to be able to find the minimum score. For example, given $S_1 = AC$ and $S_2 = CA$, a possible break is at (1, 1), this gives four parts of the sequences, namely, $S_1[1, 1] = A$, $S_1[2, 2] = C$, $S_2[1, 1] = C$, and $S_2[2, 2] = A$. Any of the parts can be aligned to one another, and since the scores of aligning those parts are already calculated, since the parts are all of shorter lengths, then it is easy to compare the scores resulting from different combinations. The optimal score is obtained by aligning $S_1[1, 1]$ to $S_2[2, 2]$, and aligning $S_1[2, 2]$ to $S_2[1, 1]$, this gives a score of $0 \cdot w_m$, since all values are identical, plus a break penalty w_b. For longer subsequences, more breaks are considered, and all are processed to find the minimum score. This minimum score is only recorded if it is less than aligning the two subsequences without any breaks, otherwise, a score without breaks is recorded in the AlignScore array.

Therefore the base cases are, for $1 \leq i \leq m$ and $1 \leq j \leq n$, as follows:

$$AlignScore[i,i,j,j] = \begin{cases} 0 & \text{if } S_1[i,i] = S_2[j,j], \\ w_m & \text{otherwise.} \end{cases}$$

Table 1. Scoring Function Used by RegAlign

$s(x, y)$	A	C	G	T	—
A	0	4	4	4	4
C	4	0	4	4	4
G	4	4	0	4	4
T	4	4	4	0	4
—	4	4	4	4	0

The function RegAlign used in the following recursive relation is the classic global alignment algorithm for the case when there are only mutations, insertions, and deletions. The algorithm for RegAlign is illustrated in Figure 4, and the scoring function used by RegAlign is shown in Table 1. The algorithm for whole genome alignment using a dynamic programming approach is illustrated in Figure 5.

And, the general case:

$$AlignScore[i,k,j,l] = \min_{i \le x \le k,\, j \le y \le l} \begin{cases} RegAlign(S_1[i,k], S_2[j,l]), \\ AlignScore[i,x,j,y] + \\ \quad AlignScore[x+1,k,y+1,l] + w_b, \\ AlignScore[x+1,k,j,y] + \\ \quad AlignScore[i,x,y+1,l] + w_b \end{cases}$$

AlignScore $[i, k, j, l]$ fetches a value from the AlignScore array.

Illustrative Example. This example shows the steps of the dynamic programming algorithm, Table 2 shows the alignment scores of pairs of subsequences. Let $w_b = 1$, hence $w_m = w_f = w_d = 4 \times w_b = 4$, and let the empty sequence be λ. $S_1 = AC$ and $S_2 = CA$.

Theorem 2. *Algorithm TrueAlign has time complexity* $O(m^4 n^4)$.

Proof. From the pseudocode in Figure 5, there are six nested loops, three repeat m times and three repeating n times. The function RegAlign, shown in Figure 4 has a complexity of $O(mn)$. Therefore, the overall complexity of whole genome alignment is $O(m^4 n^4)$. ■

```
TrueAlign($S_1[1,m]$, $S_2[1,n]$)
for $i \leftarrow 1$ to $m$
    do for $j \leftarrow 1$ to $n$
        do if $S_1[i,i] = S_2[j,j]$
            then $AlignScore[i,i,j,j] \leftarrow 0$
            else $AlignScore[i,i,j,j] \leftarrow w_m$
for $h \leftarrow 1$ to $m$
    do for $v \leftarrow 1$ to $n$
        do for $i \leftarrow 1$ to $m - h$
            do for $j \leftarrow 1$ to $n - v$
                do $k \leftarrow i + h - 1$
                    $l \leftarrow j + v - 1$
                    for $x \leftarrow i$ to $k$
                    do for $y \leftarrow j$ to $l$
                        do $AlignScore[i,k,j,l] \leftarrow$
                        $\min\{RegAlign(S_1[i,k], S_2[j,l]),$
                        $AlignScore[i,x,j,y] +$
                        $AlignScore[x+1,k,y+1,l] + w_b,$
                        $AlignScore[x+1,k,j,y] +$
                        $AlignScore[i,x,y+1,l] + w_b\}$
return $AlignScore[1,m,1,n]$
```

FIG. 5. Algorithm for whole genome alignment.

TABLE 2. ILLUSTRATIVE EXAMPLE FOR GENOMEALIGN

S_1	S_2	*Score*
A	λ	4
A	C	4
A	A	0
A	CA	4
C	λ	4
C	C	0
C	A	4
C	CA	4
AC	λ	8
AC	C	4
AC	A	4
AC	CA	1
λ	C	4
λ	A	4
λ	CA	8

9. ARRANGEMENTS

The algorithm GenomeAlign in Figure 5 is proven optimal for a certain class of alignments. Given two aligned nucleotide sequences S_1 and S_2, the two sequences are broken into blocks. The blocks are separated by breaks. The blocks are numbered for S_1 and S_2, and if the sequence of numbers obtained for S_2 can be recursively broken into a prefix and a suffix of the block numbers obtained for S_1, then this alignment can be obtained by GenomeAlign.

As an example, take $S_1 = ACCCGT$ and $S_2 = CACTGC$. An alignment graph for S_1 and S_2 is shown in Figure 6. By performing the alignment between S_1 and S_2, S_1 could be broken into blocks that correspond to the parts aligned to S_2. For example, AC in S_1 are nucleotides numbers 1 and 2, and they are aligned with nucleotides 2 and 3 in S_2, this makes block number 1 in S_1 align with block number 2 in S_2. The rest of the blocks are obtained using the same approach. Therefore, the blocks corresponding to this example are $B_1 = 1, 2, 3, 4$ and $B_2 = 2, 1, 4, 3$. If B_1 is taken as a reference, then B_2 can be broken into a prefix and a suffix of B_1. Therefore, GenomeAlign solves optimally this alignment instance.

Another example is when $S_1 = ACCCGT$ and $S_2 = GCCATC$. An alignment graph for S_1 and S_2 is shown in Figure 7. In this example, aligning S_1 and S_2 results in nucleotide 1 in S_1 being aligned with nucleotide 4 in S_2, this results in one block. Then nucleotides 2 and 3 in S_1 are also 2 and 3 in S_2, this is another block. And so on, the blocks are numbered with reference to S_1. Therefore, the blocks corresponding to this example are $B_1 = 1, 2, 3, 4, 5$ and $B_2 = 4, 2, 1, 5, 3$. Taking B_1 as a reference, B_2 cannot be broken into a prefix and a suffix of B_1, and hence it is not solved by GenomeAlign.

Let S be a set of n integers. An *arrangement* A of S is a sequence of integers of length n in which every element of S occurs exactly once. For example, if $S = \{1, 4, 5, 7, 12\}$, then 5, 12, 1, 7, 4 is an arrangement of S, but neither 4, 1, 3, 7, 5 nor 12, 7, 4, 5, 4 is. The *reverse* of A is denoted by A^R. If A is an arrangement, then $A(i)$ is the element in position i in the sequence. For example, if $A = 5, 12, 1, 7, 4$, then $A(4) = 7$. If $n \geq 1$ is an integer, then the *identity arrangement* for n is the sequence $I_n = 1, 2, \ldots, n$. Let S be any set of n integers. Then sort(S), the *sorted arrangement* of S, is the unique arrangement of S in which the elements

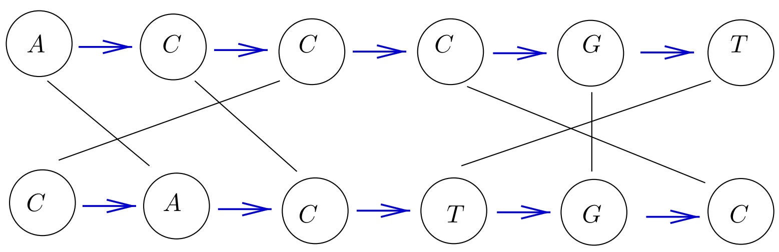

FIG. 6. An alignment that can be obtained by GenomeAlign.

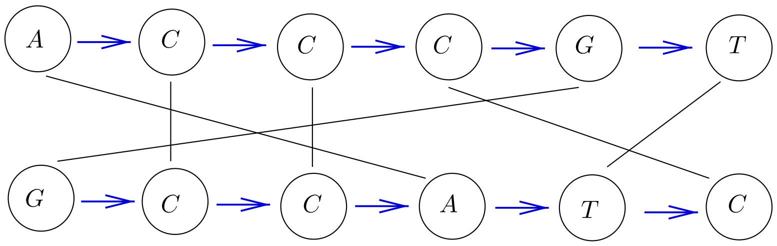

FIG. 7. An alignment that cannot be obtained by GenomeAlign.

appear in increasing order. For example, sort({1, 4, 5, 7, 12}) $= 1, 4, 5, 7, 12$ and sort({1, 2, 3, 4, 5}) $= I_5$. If A is an arrangement of n elements, then the *size* of A is size(A) $= n$.

If $A = a_1, a_2, \ldots, a_n$ is an arrangement and $1 \le i \le j \le n$, then $A[i, j] = a_i, a_{i+1}, \ldots, a_j$ is a *subarrangement* of A; of course, $A[i,j]$ is an arrangement of the set $\{a_i, a_{i+1}, \ldots, a_j\}$. For example, given an arrangement $A = 5, 7, 6, 4, 9, 8$, then the following is a subarrangement: $A[3, 5] = 6, 4, 9$. An arrangement A is *consecutive* if it is a subarrangement of I_n, for some n. A set S is *consecutive* if sort(S) is consecutive. For example, the arrangement 3, 4, 5, 6, 7, 8 is consecutive, while neither 4, 3, 5, 6 nor 3, 4, 7, 8 is. The set {2, 4, 3, 6, 5} is consecutive, but the set {2, 4, 6, 5} is not.

The class of *breakable arrangements* is defined recursively, as follows. The base cases are subarrangements of I_n for some n, which are all breakable arrangements. Now, assume that A_1 is a breakable arrangement of a consecutive set S_1 and that A_2 is a breakable arrangement of a consecutive set S_2 such that $S_1 \cap S_2 = \emptyset$ and such that $S_1 \cup S_2$ is consecutive. Then, both A_1A_2 and A_2A_1 are breakable arrangements. For example, if $A_1 = 6, 4, 5$ and $A_2 = 2, 3, 1$, then it is easy to show that A_1 and A_2 are breakable. Moreover, {6, 4, 5}, {2, 3, 1}, and {6, 4, 5, 2, 3, 1} are all consecutive sets. Hence, 6, 4, 5, 2, 3, 1 and 2, 3, 1, 6, 4, 5 are breakable arrangements. However, 6, 4, 5 cannot be combined with 2, 1 to form a breakable arrangement, since {6, 4, 5, 2, 1} is not a consecutive set.

The definition of a breakable arrangement implies that, for every breakable arrangement A, there exists a binary tree T with A at the root, breakable subarrangements of A at every node, and subarrangements of I_n at the leaves. Such a tree is called a *break tree* for the arrangement. For example, consider $A = 2, 1, 7, 5, 6, 8, 9, 10, 4, 3$. Starting from the left of A, the first place that we can break yields subarrangements 2, 1 and 7, 5, 6, 8, 9, 10, 4, 3. The subarrangement 2, 1 breaks further into the two subarrangements 2 and 1 of an identity arrangement. The subarrangement 7, 5, 6, 8, 9, 10, 4, 3 breaks into 7, 5, 6, 8, 9, 10 and 4, 3. Continuing recursively, we get the break tree in Figure 8.

An example of a non-breakable arrangement is $A = 2, 1, 7, 5, 3, 8, 10, 9, 4, 6$. The first division into consecutive subarrangements is 2, 1 and 7, 5, 3, 8, 10, 9, 4, 6. However, it is not possible to further break 7, 5, 3, 8, 10, 9, 4, 6 into two consecutive subarrangements.

Theorem 3. *The two shortest non-breakable arrangements are* $i+2, i, i+3, i+1$ *and* $i+1, i+3, i, i+2$.

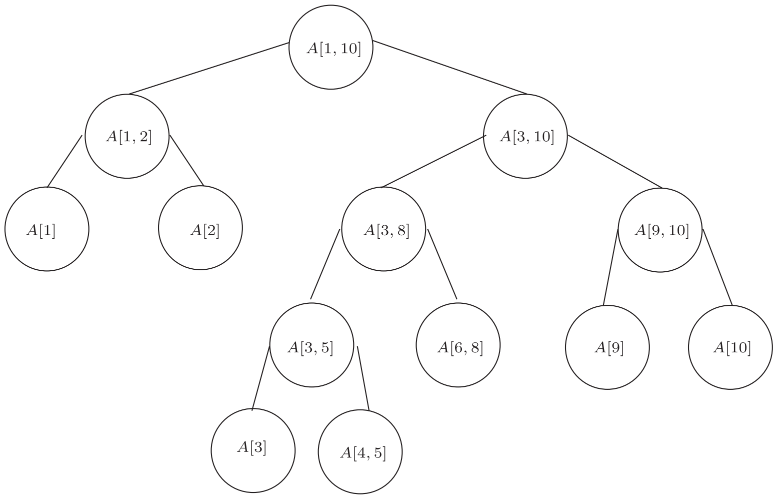

FIG. 8. A break tree for $A = 2, 1, 7, 5, 6, 8, 9, 10, 4, 3$.

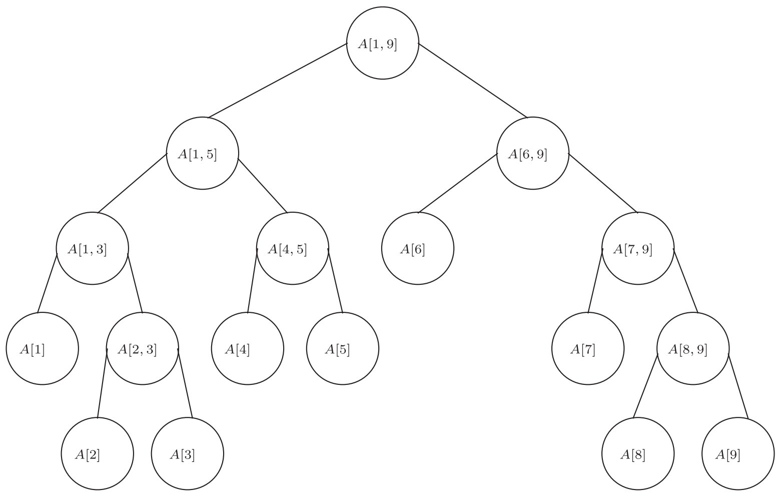

FIG. 9. Break tree for $A = 7, 6, 5, 9, 8, 4, 3, 2, 1$.

Proof. From the definition of breakable arrangements, we know that a breakable arrangement must be able to break into a prefix and a suffix of the identity arrangement I_n. For the arrangement $i+2, i, i+3, i+1$, the possible breaks are $i+2$ and $i, i+3, i+1$; $i+2, i$ and $i+3, i+1$; and $i+2, i, i+3$ and $i+1$. In all three cases, there is an arrangement that is not consecutive. Therefore, $i+2, i, i+3, i+1$ is not breakable. The same argument works for $i+1, i+3, i, i+2$. ■

Theorem 4. *If A is a breakable arrangement, then its reverse A^R is also breakable.*

Proof. Given a breakable arrangement A, we know that A can be recursively broken into a prefix and a suffix of the identity arrangement I corresponding to A. Therefore, it is clear that A^R can similarly be broken in the same way, by simply recursively reversing the break tree obtained for A. ■

10. IDENTIFYING BREAKABLE ARRANGEMENTS

In this section, we address the identification problem for breakable arrangements:

Identify Breakable Arrangement
INSTANCE: Arrangement A.
QUESTION: Is A breakable?

If A is breakable, the algorithm should construct a break tree for A. We construct the break tree top-down.

Using the recursive definition of a breakable arrangement, at any stage we have a consecutive arrangement A that we need to represent as A_1A_2, where A_1 and A_2 are also consecutive arrangements. For example, given the consecutive arrangement $A = 7, 3, 9, 5, 8, 4, 2, 1, 6$, the algorithm identifies two consecutive subarrangements $A[1, i]$ and $A[i+1, n]$, where $n = \text{size}(A) = 9$ and i is as small as possible. If we consider $i = 1$, we have $A[1, 1] = 7$ and $A[2, 9] = 3, 9, 5, 8, 4, 2, 1, 6$; we see that $A[2, 9]$ is not a consecutive arrangement. If we consider $i = 2$, we have $A[1, 2] = 7, 3$ and $A[3, 9] = 9, 5, 8, 4, 2, 1, 6$; in this case, neither subarrangement is consecutive. As we consider all other values of i, we see that at least one of the corresponding two subarrangements is not consecutive. We conclude that A is not breakable.

Now consider an example of a breakable arrangement. Let $A = 7, 6, 5, 9, 8, 4, 3, 2, 1$. A can be divided into $A[1, 5]$ and $A[6, 9]$, which are consecutive subarrangements of A. Hence, a top-down algorithm can address each of $A[1, 5]$ and $A[6, 9]$ recursively. Continuing recursively, we ultimately end up with the break tree in Figure 9.

Figure 10 shows the algorithm for solving Identify Breakable Arrangement.

Theorem 5. *The algorithm Breakable has time complexity $O(n^3 \log n)$.*

Proof. Each call to the recursive algorithm Breakable is performed in time $Cn^2 \log n$, for some constant $C > 0$. The loop on line 8 repeats n times, and in each iteration sorting is performed, and this takes time

```
Breakable($A, N$)
if $A$ is a subarrangement of an identity arrangement
    then return TRUE
$n \leftarrow$ size($A$)
$B \leftarrow$ sort($A$)
if $B$ is not consecutive
    then return FALSE
for $i \leftarrow 1$ to $n - 1$
    do $A_1 \leftarrow A[1, i]$
        $A_2 \leftarrow A[i + 1, n]$
        $B_1 \leftarrow$ sort($A_1$)
        $B_2 \leftarrow$ sort($A_2$)
        if $B_1$ is consecutive and $B_2$ is consecutive
            then Add child $N_1$ labeled $A_1$
                    with parent $N$
                Add child $N_2$ labeled $A_2$
                    with parent $N$
                return Breakable($A_1, N_1$) and
                        Breakable($A_2, N_2$)
return FALSE
```

FIG. 10. Algorithm for identifying breakable arrangements.

$Cn\log n$. Therefore, the time needed for the loop is $Cn^2 \log n$. The function is recursively called for subarrangements of the input A. The worst case for the sizes of the subarrangements $A[1, i]$ and $A[i+1, n]$ is when $A[1, i]$ has a size of 1. Therefore, the complexity of the function can be expressed using the following recurrence:

$$T(1) = 1$$
$$T(2) = 2$$
$$T(n) - T(1) + T(n-1) + Cn^2 \log n.$$

From the above recurrence, it is seen that, in the worst case, the function is called n times. Hence, the recursive algorithm Breakable is of complexity $O(n^3 \log n)$. ■

Theorem 6. *The algorithm Breakable identifies all breakable arrangements and only breakable arrangements.*

Proof. If Breakable(A, r) returns TRUE, then it has built a break tree with root r that proves that A is breakable. Hence, Breakable(A, r) returns TRUE only for breakable arrangements.

To prove that algorithm Breakable identifies all breakable arrangements, we proceed by induction on the size of the input arrangement. The base case occurs for arrangements of size 1. Such arrangements are necessarily subarrangements of an identity arrangement and hence are breakable. In this case, Algorithm Breakable returns TRUE, as required.

The inductive hypothesis is that, for $n \geq 1$, all breakable arrangements of size $\leq n$ are identified correctly by algorithm Breakable.

Now, assume that A is a breakable arrangement of size $n+1$. Without loss of generality, assume that A is an arrangement of the set $1, 2, \ldots, n+1$. Let T be a break tree for A, and let $A[1, j]$ and $A[j+1, n+1]$ be the labels of the children of the root of T. Furthermore, let i be the value chosen by the algorithm Breakable so that $A[1, i]$ and $A[i+1, n+1]$ are selected as the labels of N_1 and N_2 in the first recursive call of Breakable. Clearly, $i \leq j$. If $i = j$, then $A[1, i]$ and $A[i+1, n+1]$ are both breakable and the inductive hypothesis implies that Breakable will correctly identify them and hence A as breakable. If $i < j$, then we proceed as follows.

Without loss of generality, assume that every element of $A[1, i]$ is less than every element of $A[i+1, n+1]$ (the remaining case that every element of $A[1, i]$ is greater than every element of $A[i+1, n+1]$ is symmetric). Then 1 is an entry of $A[1, i]$, and $n+1$ is an entry of $A[i+1, n+1]$. We claim that $A[1, i]$ is the label of some node in T. To see this, consider the node labels of the nodes on the leftmost path of T. Without loss of generality, we can write the labels in order as $A[1, s_1], A[1, s_2], \ldots, A[1, s_t]$, where $s_1 = n+1$, $s_2 = j$, and $A[1, s_t]$ is a subarrangement of I_{n+1}. Of necessity, $s_1 > s_2 > \cdots > s_t$. Hence, we can select a unique k such that $s_k > i$ and $s_{k+1} \leq i$. If $s_{k+1} = i$, then the claim that $A[1, i]$ is the label of some node in T is true. Otherwise, to obtain a contradiction, assume that $s_{k+1} < i$. We have that $A[1, s_{k+1}]$ and $A[s_{k+1}+1, s_k]$ are both breakable, since they are labels in T. Since every element of $A[s_{k+1}+1, i]$ is less than every element of $A[i+1, s_k]$, both $A[s_{k+1}+1, i]$ and $A[i+1, s_k]$ must be consecutive. $A[1, s_{k+1}]$ cannot contain the element 1, since then algorithm Breakable would have selected s_{k+1} instead of i. So $A[s_{k+1}, i]$ must contain the element 1. Since it is breakable, $A[s_{k+1}, i]$ must in fact contain the elements $1, 2, \ldots, s_k - s_{k+1}$. However, every element of $A[1, s_{k+1}]$ is smaller than every element of $S[i+1, s_k]$, which gives us the desired contradiction. We conclude that $A[1, i]$ is the label of some node in T and hence breakable. Moreover, since $A[1, i]$ is on the leftmost path of T, we have that $A[i+1, n+1]$ is breakable as well. By the inductive hypothesis, the recursive calls by Breakable correctly identify $A[1, i]$ and $A[i+1, n+1]$ as breakable. Hence, Breakable identifies A as breakable. By induction, we conclude that algorithm Breakable identifies all breakable arrangements. ∎

11. COUNTING BREAKABLE ARRANGEMENTS

For an integer $n \geq 1$, let $C(n)$ be the number of breakable arrangements on the set $\{1, 2, \ldots, n\}$. To compute $C(n)$, we proceed as follows. The base cases are $C(1) = 1$ and $C(2) = 2$. Let $n \geq 3$. Let A be a breakable arrangement of $\{1, 2, \ldots, n\}$. By the definition of breakable arrangements, there exists an i such that $1 \leq i \leq n-1$ and such that there is a breakable arrangement A_1 of $\{1, 2, \ldots, i\}$ and a breakable arrangement A_2 of $\{i+1, i+2, \ldots, n\}$ satisfying $A = A_1A_2$ or $A = A_2A_1$. For a fixed i, there are $2C(i)C(n-i)$ ways to create a breakable arrangement of $\{1, 2, \ldots, n\}$. Hence,

$$C(n) \leq \sum_{i=1}^{n-1} 2C(i)C(n-1).$$

It is not equal because, for $i \geq 2$, some of the $2C(i)C(n-i)$ arrangements for i may be counted in arrangements for some i', where $i' < i$. For example, the breakable arrangement 2, 1, 3, 4 will be counted both for $i = 2$ and for $i = 3$.

We claim that exactly half of the $2C(i)C(n-i)$ arrangements for i, where $i \geq 2$, are counted earlier. Fix $i \geq 2$. Let A_1 be a breakable arrangement of $\{1, 2, \ldots, i\}$, and let A_2 be a breakable arrangement of $\{i+1, i+2, \ldots, n\}$. Let A_1^R be the reverse of A_1; it is easy to see that A_1^R is also breakable. We claim that precisely one of A_1A_2 and $A_1^RA_2$ is counted by an i' with $i' < i$. (The argument for A_2A_1 and $A_2A_1^R$ is similar.) Without loss of generality, we may assume that i appears before 1 in A_1. Let $A_1 = a_1, a_2, \ldots, a_i$, let $a_j = i$, and let $a_k = 1$. Then $1 \leq j < k \leq i$. Since A_1 is breakable, there exists m, where $j \leq m < k$, such that $A_1[m+1, i]$ is a breakable arrangement of $\{1, 2, \ldots, i-m\}$, and $A_1[1, m]$ is a breakable arrangement of $\{i-m+1, i-m+2, \ldots, i\}$. Then, $A_1^R[1, i-m]$ is a breakable arrangement of $\{1, 2, \ldots, i-m\}$, and $A_1^R[i-m+1, i]$ is a breakable arrangement of $\{i-m+1, i-m+2, \ldots, i\}$. Because i appears before 1 in A_1, A_1A_2 was not counted earlier. Because $A_1^R[1, i-m], A_1^R[i-m+1, i]$, and A_2 are breakable, we have that $A_1^R[i-m+1, i]A_2$ is breakable and that $A_1^RA_2$ is counted earlier. Hence, exactly half of the $2C(i)C(n-i)$ arrangements for i, where $i \geq 2$, are counted earlier.

We arrive at the following recurrence:

$$
\begin{aligned}
&C(1) = 1\\
&C(2) = 2\\
&C(n) = 2C(1)C(n-1) + \sum_{i=2}^{n-1} C(i)C(n-1),
\end{aligned}
$$

where $n \geq 3$.

Figure 11 shows the algorithm for counting breakable arrangements.

Theorem 7. *The algorithm CountBreakArrange has time complexity $O(n^2)$.*

Proof. There are two nested loops on lines 4 through 6, the loop on line 4 repeats n times, and the loop on line 6 repeats n times, in the worst case. The statement inside the nested loops, on line 7, is of constant time, as it performs access on calculated values. Therefore, the overall complexity of the algorithm is $O(n^2)$. ■

The sequence of counts is the same sequence as the Schröder numbers (Shapiro and Stephens, 1991), which can be expressed by the following recurrence (Weisstein, 2003):

$$
\begin{aligned}
&S_0 = 1\\
&S_1 = 2\\
&S_n = S_{n-1} + \sum_{i=0}^{n-1} S_i S_{n-1-i},
\end{aligned}
$$

where $n \geq 2$. The equivalence of the sequences is embodied in the following theorem.

Theorem 8. *For $n \geq 0$, $C(n+1) = S_n$.*

Proof. The theorem is immediately true for $n = 0$ and $n = 1$. We proceed by induction on n, $n \geq 2$, assuming that the theorem is true for numbers smaller than n and showing that $C(n+1) = S_n$. Fix $n \geq 2$. Using the inductive hypothesis, we have

$$
\begin{aligned}
C(n+1) &= 2C(1)C(n) + \sum_{i=2}^{n} C(i)C(n+1-i)\\
&= 2S_0 S_{n-1} + \sum_{i=2}^{n} S_{i-1} S_{n-i}\\
&= 2S_0 S_{n-1} + \sum_{i=1}^{n-1} S_i S_{n-1-i}
\end{aligned}
$$

```
1  CountBreakArrange($n$)
2  $Count[1] \leftarrow 1$
3  $Count[2] \leftarrow 2$
4  for $i \leftarrow 3$ to $n$
5        do $Count[i] \leftarrow 2Count[1]Count[i-1]$
6             for $j \leftarrow 2$ to $i-1$
7                  do $Count[i] \leftarrow Count[i] +$
                         $Count[j]Count[i-j]$
```

FIG. 11. Algorithm for counting breakable arrangements.

$$
\begin{aligned}
&= S_0 S_{n-1} + \sum_{i=0}^{n-1} S_i S_{n-1-i} \\
&= S_{n-1} + \sum_{i=0}^{n-1} S_i S_{n-1-i} \\
&= S_n.
\end{aligned}
$$

By induction, the theorem holds for all n. ■

12. DYNAMIC PROGRAMMING AND ARRANGEMENTS

In this section, we prove that the dynamic programming algorithm presented in Section 8 obtains optimal results for breakable arrangements.

Theorem 9. *The algorithm GenomeAlign yields optimal results for breakable arrangements.*

Proof. By contradiction. Suppose there are two sequences S_1 and S_2. S_1 can be broken into blocks such that each block is numbered to obtain the identity arrangement $\text{In} = 1, 2, 3, \ldots, n$. Suppose S_2 is breakable, that is, S_2 can be broken into blocks that can be numbered then mapped to the blocks of S_1. The blocks of S_2 are represented by the breakable arrangement R_2. Assume S_1 and S_2 can not be optimally aligned by TrueAlign. From the definition of breakable arrangements, it is known that R_2 can be divided into subsets that are either of size one, or are blocks of the identity arrangement. If TrueAlign can not optimally align S_1 and S_2, then TrueAlign can not optimally align sequences of length one, or identical sequences (blocks of the identity arrangement), however, those are trivial cases solved by TrueAlign, and hence this gives a contradiction. Therefore, the theorem is true. ■

13. ARRANGEMENTS AND OTHER ALIGNMENTS

In this section, we study some alignment examples from the literature and see how they map to the arrangements we define. For each alignment, the blocks are identified and numbered to obtain arrangements. One of the arrangements is taken as a reference, and the breakability of the other arrangement is tested accordingly.

The first alignment we consider is shown in Figure 6 of Darling et al. (2004). This alignment is obtained for nine genomes listed in Table 1 of Darling et al. (2004); these are E. coli K12 MG1655, E. coli O157:H7 EDL933, E. coli O157:H7 VT-2 Sakai, E. coli CFT073, S. flexneri 2A 2457T, S. flexneri 2A, S. enterica Typhimurium LT2, S. enterica Typhi CT18, and S. enterica Typhi Ty2. The alignment was done using the MAUVE alignment tool (Darling et al. 2004). The alignment shown in Figure 6 of Darling et al. (2004) is color coded, where the corresponding blocks in different genomes have the same color, and the numbers for the blocks were obtained accordingly. The topmost row in the figure is taken as a reference, where 25 blocks were identified, giving the reference identity arrangement $I = 1, 2, 3, \ldots, 25$. The other eight arrangements were obtained in a similar manner, by mapping the color-codes to the first genome. The arrangements obtained are as follows:

$$
\begin{aligned}
A_1 &= 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \\
&\quad 18, 19, 20, 21, 22, 23, 24, 25 \\
A_2 &= 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \\
&\quad 18, 19, 20, 21, 22, 23, 24, 25 \\
A_3 &= 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \\
&\quad 18, 19, 20, 21, 22, 23, 24, 25 \\
A_4 &= 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \\
&\quad 18, 19, 20, 21, 22, 23, 24, 25
\end{aligned}
$$

$$A_5 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 21, 22, 20, 19, 18, 17, 16, 24, 23, 25$$

$$A_6 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 21, 24, 23, 25$$

$$A_7 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25$$

$$A_8 = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 19, 20, 18, 17, 21, 22, 23, 24, 25$$

$$A_9 = 1, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 16, 19, 20, 18, 17, 21, 22, 23, 24, 25$$

All these arrangements are breakable.

MUMmer of Delcher et al. (1999) is used to align M. genitalium and M. pneumoniae. Figure 7 (bottom) (Delcher et al., 1999) gives the identity arrangement $I_7 = 1, 2, 3, 4, 5, 6, 7$ for the sequence represented on the x-axis, and $A = 7, 2, 1, 3, 4, 5, 6$ for the sequence represented on the y-axis. It is easy to check that A is a breakable arrangement.

In Joseph and Sasikumar (2006), chaos game representation (CGR) of sequences is used to obtain regions of similarity between two input sequences. The CGR representation of a sequence is a dot-matrix plot, where the points on the plot are mathematically calculated. Figure 3 of Joseph and Sasikumar (2006) yields two identical arrangements with duplication for the genomes HIV type 1 and CIV, so that is clearly breakable. Also, for Figure 4 of Joseph and Sasikumar (2006), we obtain the identity arrangement $I_8 = 1, 2, 3, 4, 5, 6, 7, 8$ for the sequence on the x-axis, Pyrococcus abyssi GE5, and for that on the y-axis, Pyrococcus horikoshii, we get the arrangement $A = 1, 7, 5, 6, 4, 3, 2, 8$, which is breakable.

In Kurtz et al. (2004), MUMmer is used to align A. fumigatus and A. nidulans. The arrangements $I_{10} = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ and $A = 10, 5, 6, 7, 8, 9, 4, 3, 2, 1$ are obtained for the x-axis and y-axis, respectively, in Figure 1 of Kurtz et al. (2004). A is clearly breakable.

Other alignments from the literature also yield breakable arrangements. For example, Figure 2A of Blanchette et al. (2004) shows an alignment between the chloroplast genomes of Arabidopsis thaliana and Oenothera elata using threaded blockset alignment (TBA). This alignment yields two identical arrangements, with duplication. Also, Figure 5 of Lu et al. (2006) shows an alignment obtained for EBV and EHV2 using GenomeBlast (Lu et al., 2006), and it is clearly breakable.

The alignment tool CoCoNUT (Abouelhoda et al., 2008) is used to align orthologs of human chromosome X to mouse chromosome X. In Figure 5 of Abouelhoda et al. (2008), the blocks on the solid lines on the x-axis (human) are mapped to the identity arrangement $I_5 = 1, 2, 3, 4, 5$. However, the corresponding blocks on the y-axis (mouse) yield the arrangement $A = 3, 5, 2, 4, 1$, which is not breakable.

Therefore, it is seen that all but one of the real world examples analyzed are breakable arrangements that can be optimally solved by GenomeAlign.

14. A GREEDY ALGORITHM FOR WHOLE GENOME ALIGNMENT

A greedy approach can be used to obtain a heuristic to solve the whole genome alignment problem. This greedy approach takes as input two sequences. Let the two sequences be denoted by S_1 and S_2, and let the complement of S_2 be $S_2{}^c$. The algorithm in Figure 12 consists of three steps, the first is the preprocessing step, the second is the alignment step, and the third step is the scoring step.

The preprocessing step in Figure 13 finds the alignment score between each pair of subsequences of S_1 and each of S_2 and $S_2{}^c$, and the values are kept in a Preprocesstable, where there is an attribute in the table to indicate whether S_2 is complemented or not. An alternative method for the preprocessing step, shown in Figure 14, is to have a prespecified subsequence size, k. This will reduce the amount of preprocessing to only a subset of all pairs of subsequences. The alignment step in Figure 15 starts processing the two subsequences, this is done by greedily aligning pairs of subsequences according to their alignment score, computed in the preprocessing step. The pairs are processed in ascending order of their score. This step continues until the two sequences are finished. If the pair being processed already contains a subsequence

```
GreedyAlign($S_1$, $S_2$)
PreprocessPairs($S_1$, $S_2$) ▷ Or alternatively, ReducedPreprocessPairs($S_1, S_2, k$)
Align($S_1$, $S_2$)
Score($G$)
```

FIG. 12. The greedy algorithm GreedyAlign.

```
PreprocessPairs($S_1$, $S_2$)
for $i \leftarrow 1$ to $m$
do for $k \leftarrow i$ to $m$
    do for $j \leftarrow 1$ to $n$
        do for $l \leftarrow j$ to $n$
            do $PreprocessTable[i, k, j, l] \leftarrow$ RegAlign($S_1[i, k]$, $S_2[j, l]$)
            $PreprocessTable[i, k, j, l].cmpl \leftarrow 0$
            $PreprocessTable[i', k', j', l'] \leftarrow$ RegAlign($S_1[i, k]$, $S_2{}^c[j, l]$)
            $PreprocessTable[i', k', j', l'].cmpl \leftarrow 1$
```

FIG. 13. The preprocessing step.

that has been aligned in a previous step, then this indicates a duplication. The scoring step in Figure 16 then follows to compute the score of the alignment done in step two.

For example, let $S_1 = AC$, $S_2 = CA$, and $S_2{}^c = GT$, the preprocessing step will produce Table 3.

The alignment step then sorts the scores and returns the alignment shown in Figure 17.

This alignment is then scored, in the final step.

Theorem 10. *Let S_1 be a sequence of length m and S_2 be a sequence of length n. Then GreedyAlign produces a whole genome alignment of S_1 and S_2 in $O(m^3n^3)$ time.*

```
ReducedPreprocessPairs($S_1$, $S_2$, $k$)
for $i \leftarrow 1$ to $(m/k) + 1$
do for $j \leftarrow 1$ to $(n/k) + 1$
    do $PreprocessTable[i, i + k, j, j + k] \leftarrow$ RegAlign($S_1[i, i + k]$, $S_2[j, j + k]$)
    $PreprocessTable[i, i + k, j, j + k].cmpl \leftarrow 0$
    $PreprocessTable[i', i' + k, j', j' + k] \leftarrow$ RegAlign($S_1[i, i + k]$, $S_2{}^c[j, j + k]$)
    $PreprocessTable[i', i' + k, j', j' + k].cmpl \leftarrow 1$
```

FIG. 14. The preprocessing step with specified subsequence size.

```
1  Align($S_1$, $S_2$)
2  Sort(PreprocessTable)
3  $count \leftarrow 0$
4  while ($S_1 \neq \lambda$ or $S_2 \neq \lambda$ or ${S_2}^c \neq \lambda$)
5  do $(i, k, j, l) \leftarrow Preprocess[count]$
6      if $PreprocessTable[i, j, k, l].cmpl = 0$
7      then if $S_1[i, k]$ and $S_2[j, l]$ not aligned
8          then $S_1 \leftarrow S_1 - S_1[i, k]$
9              $S_2 \leftarrow S_2 - S_2[j, l]$
10             Align $S_1[i, k]$ and $S_2[j, l]$
                   by PreprocessTable$[i, j, k, l]$
11             $count \leftarrow count + 1$
12         else if $S_1[i, k]$ and $S_2[j, l]$ not aligned
13             then $S_1 \leftarrow S_1 - S_1[i, k]$
14             ${S_2}^c \leftarrow {S_2}^c$-${S_2}^c[j, l]$
15             Align $S_1[i, k]$ and ${S_2}^c[j, l]$
                   by Preprocess$[i, j, k, l]$
16             $count \leftarrow count + 1$
```

FIG. 15. The alignment step.

Proof. The preprocessing step contains four nested loops, two of which repeat m times, and the other two repeat n times, this gives time complexity $O(m^2n^2)$. The step done inside the inner loop, which is the RegAlign has time complexity $O(mn)$. Therefore, the preprocessing step has time complexity $O(m^3n^3)$. The alignment step first sorts the Preprocesstable, this takes $O(m^2n^2 \log m^2n^2)$. Then the alignment step iterates on S_1 and S_2. The number of iterations is either m or n. Therefore, this loop is $O(m)$, if $m > n$, or $O(n)$, otherwise. Therefore, the complexity of the alignment step is $O(m^2n^2 \log m^2n^2)$. The final scoring step loops on the nodes of the alignment graph and scores it. This is done in $O(mn)$. Therefore, the complexity of this greedy approach is $O(m^3n^3)$. ■

Theorem 11. *Let S_1 be a sequence of length m, S_2 be a sequence of length n, and k be the length of subsequences. Then GreedyAlign using ReducedPreprocessPairs produces a whole genome alignment of S_1 and S_2 in* $\max\{O(k^2mn), O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)\}$ *time.*

Proof. Without loss of generality, assume that $m \geq n$. The preprocessing step in Figure 14 contains two nested loops, each repeating k times, giving time complexity $O(k^2)$. The step done inside the inner loop, which is the RegAlign, has time complexity $O(mn)$. Therefore, the preprocessing step has time complexity $O(k^2mn)$. The alignment step first sorts the Preprocesstable, this takes $O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)$. Then the alignment step iterates on S_1 and S_2. The number of iterations is either m or n. Therefore, this loop is $O(m)$. Therefore, the complexity of the alignment step is $O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)$. The final scoring step loops on the nodes of the alignment graph and scores it. This is done in $O(mn)$. Therefore, the complexity of this greedy approach is $\max\{O(k^2mn), O((m/k)^2(n/k)^2 \log(m/k)^2(n/k)^2)\}$. ■

Example 1. This example shows the steps of the greedy algorithm in detecting reversals. Table 4 shows the alignment scores of pairs of subsequences. Let $w_b = 1$, and let $w_m = w_f = w_d = 4$. Starting by the lowest scores, of value 0, it is seen that the lower scores come from aligning S_1 with ${S_2}^c$. $S_1[1, 1]$ aligns with

```
1  Score($G(V, U, E_{\text{blue}}, E_{\text{black}}, E_{\text{red}})$)
2  $m \leftarrow 0$
3  $b \leftarrow 0$
4  $f \leftarrow 0$
5  $currentV \leftarrow 0$
6  $currentU \leftarrow 0$
7  for $i = 1$ to $|V|$
8      do if $v_i$ is not connected to $currentU + 1$
9             then $b \leftarrow b + 1$
10         if $v_i$ OR $u_{currentU}$ are not aligned
11            then $f \leftarrow f + 1$
12         if $e_{v_i u_{currentU}} \in E_{\text{red}}$ and
                 $Label(v_i) \neq Label^c(u_{currentU})$
                 or $e_{v_i u_{currentU}} \in E_{\text{black}}$ and
                 $Label(v_i) \neq Label(u_{currentU})$
13            then $m \leftarrow m + 1$
14     $currentU \leftarrow currentU + 1$
15     $currentV \leftarrow currentV + 1$
16 $score \leftarrow mw_m + fw_f + bw_b$
17 return $score$
```

FIG. 16. The scoring step.

${S_2}^c[2, 2]$ using a red edge. Similarly, $S_1[2, 2]$ aligns with ${S_2}^c[1, 1]$ using a red edge. This gives a score of $w_b = 1$.

$$S_1 = AC$$
$$S_2 = GT$$
$${S_2}^c = CA$$

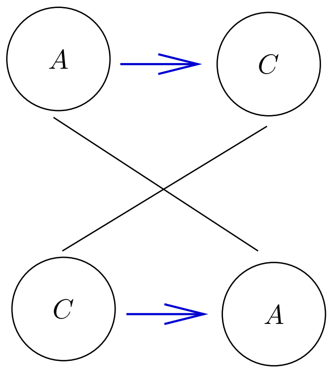

FIG. 17. Alignment graph for *AC* and *CA*.

Table 3. Output of the Preprocessing Step for $S_1 = AC$ and $S_2 = CA$

S_1	S_2 or S_2^c	Complement (1/0)	Score
A	λ	0	4
A	C	0	4
A	A	0	0
A	CA	0	4
C	λ	0	4
C	C	0	0
C	A	0	4
C	CA	0	4
AC	λ	0	8
AC	C	0	4
AC	A	0	4
AC	CA	0	8
λ	C	0	4
λ	A	0	4
λ	CA	0	8
A	G	1	4
A	T	1	4
A	GT	1	8
C	G	1	4
C	T	1	4
C	GT	1	8
AC	G	1	8
AC	T	1	8
AC	GT	1	8
λ	G	1	4
λ	T	1	4
λ	GT	1	8

The presented greedy algorithm fails to find the optimal alignment for sequences that have long common subsequences that contain a few mutations. The greedy algorithm will choose to align the mutated nucleotides with other matching nucleotides, thus, creating more breaks rather than creating mutations. For example, let $S_1 = ACCGTAGTG$ and $S_2 = TGCGACGAC$, three possible alignments are shown in Figures 18, 19, and 20.

Table 4. Greedy Algorithm: Example 1

S_1	S_2	Score	S_2^c	Score
A	λ	4		
A	G	4	C	4
A	T	4	A	0
A	GT	8	CA	4
C	λ	4		
C	G	4	C	0
C	T	4	A	4
C	GT	8	CA	4
AC	λ	8		
AC	G	8	C	4
AC	T	8	A	4
AC	GT	8	CA	8
λ	G	4		
λ	T	4		
λ	GT	8		

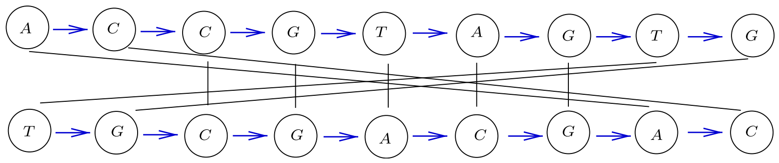

FIG. 18. Alignment graph for S_1 and S_2 with two breaks and two mutations.

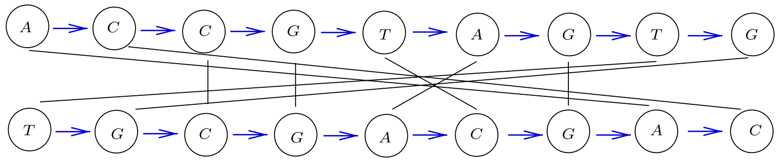

FIG. 19. Alignment graph for S_1 and S_2 with five breaks and one mutation.

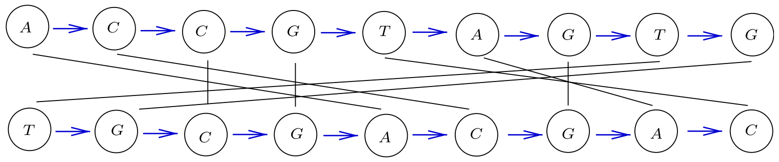

FIG. 20. Alignment graph for S_1 and S_2 with five breaks and one mutation.

The greedy algorithm will fail to obtain the alignment in Figure 18, which has the minimum number of breaks with one extra mutation.

A number of modifications can be made to the greedy algorithm to obtain better results. However, these modifications will yield a more complicated algorithm. First, for pairs of sequences with the same score calculated in the preprocessing step, longer sequences should be considered first. And, the other point is that the relative location of aligned pairs of subsequences should be taken into consideration. In other words, if a subsequence is aligned to multiple other subsequences with the same score, then the subsequence that will give better overall alignment should be chosen. For example, if one of the subsequences in S_1 is present in the same corresponding position in S_2, then this pair is better aligned than others.

15. CONCLUSION

In this article, we define alignment graphs as a model for whole genome alignment. We show that this model is capable of realizing many evolutionary events. We then define a problem that, given two DNA sequences, obtains the alignment graph with the minimum score. We also define a scoring function that we use in our algorithms. The problem defined is shown to be NP-complete, and two algorithms are presented to heuristically solve the problem. The first is a dynamic programming algorithm that is shown to obtain optimal results for breakable arrangements. The second algorithm is greedy, and it detects reversals.

In the future, experiments could be carried out to test the performance of the presented algorithms on real genomes. The weights used for different evolutionary events could be further investigated. Also, enhancements to the greedy algorithm could be made to make it detect duplications and obtain better results for the cases outlined in this article.

ACKNOWLEDGMENTS

We thank Ruth Grene, Ayman Abdel Hamid, T.M. Murali, and João Carlos Setubal for their valuable comments and Thomas Jones for implementing some of the algorithms. This work was supported by the National Science Foundation (grant ITR-0428344).

DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. 2008. CoCoNUT: an efficient system for the comparison and analysis of genomes. *BMC Bioinform.* 9, 476.

Altschul, S.F., Gish, W., Miller, W., et al. 1990. Basic local alignment search tool. *Mol. Biol.* 215, 403–410.

Blanchette, M. 2007. Computation and analysis of genomic multi-sequence alignments. *Annu. Rev. Genomics Hum. Genet.* 8, 193–213.

Blanchette, M., Kent, W.J., Riemer, C., et al. 2004. Aligning multiple genomic sequences with the Threaded Blockset Aligner. *Genome Res.* 14, 708–715.

Bray, N., Dubchak, I., and Pachter, L. 2003. AVID: a global alignment program. *Genome Res.* 13, 97–102.

Bray, N., and Pachter, L. 2004. MAVID: constrained ancestral alignment of multiple sequences. *Genome Res.* 14, 693–699.

Brudno, M., Chapman, M., Gottgens, B., et al. 2003. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinform.* 4, 66.

Brundo, M., Malde, S., Poliakov, A., et al. 2003. Glocal alignment: finding rearrangements during alignment. *Bioinformatics* 19, Suppl. 1, i54–i62.

Darling, A.C., Mau, B., Blattner, F.R., et al. 2004. MAUVE: multiple alignment of conserved genomic sequence with rearrangements. *Genome Res.* 14, 1394–1403.

Delcher, A., Kasif, S., Fleischmann, R., et al. 1999. Alignment of whole genomes. *Nucleic Acids Res.* 27, 2369–2376.

Dubchak, I., Poliakov, A., Kislyuk, A., et al. 2009. Multiple whole-genome alignments without a reference organism. *Genome Res.* 19, 682–689.

Durbin, R., Eddy, S., Krogh, A., et al. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, United Kingdom.

Ergun, F., Muthukrishnan, S., and Sahinalp, S.C. 2003. Comparing sequences with segment rearrangements. *Lect. Notes Comput. Sci.* 2914, 183–194.

Garey, M.R., and Johnson, D.S. 1979. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco.

Joseph, J., and Sasikumar, R. 2006. Chaos game representation for comparison of whole genomes. *BMC Bioinform.* 7, 243.

Junier, T., and Pagni, M. 2000. Dotlet: diagonal plots in a web browser. *Bioinformatics* 16, 178–179.

Kurtz, S., Phillippy, A., Delcher, A.L., et al. 2004. Versatile and open software for comparing large genomes. *Genome Biol.* 5, R12.

Lu, G., Jiang, L., Helikar, R.M., et al. 2006. GenomeBlast: a web tool for small genome comparison. *BMC Bioinform.* 7, S18.

Ma, J., Ratan, A., Raney, B.J., et al. 2008. The infinite sites model of genome evolution. *Proc. Nat. Acad. Sci. USA* 105, 14254–14261.

Morgenstern, B., Werner, N., Prohaska, S.J., et al. 2005. Multiple sequence alignment with user-defined constraints at GOBICS. *Bioinformatics* 21, 1271–1273.

Needleman, S.B., and Wunsch, C.D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Mol. Biol.* 48, 443–453.

Otu, H.H., and Sayood, K. 2003. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics* 19, 2122–2130.

Paten, B., Herrero, J., Fitzgerald, K.B.S., et al. 2008. Enredo and Pecan: genome-wide mammalian consistency-based multiple alignment with paralogs. *Genome Res.* 18, 1814–1828.

Pei, J., and Grishin, N. 2006. MUMMALS: multiple sequence alignment improved by using hidden Markov models with local structural information. *Nucleic Acids Res.* 34, 4364–4374.

Phuong, T.M., Do, C.B., Edgar, R.C., et at. 2006. Multiple alignment of protein sequences with repeats and rearrangements. *Nucleic Acids Res.* 34, 5932–5942.

Pohler, D., Werner, N., Steinkamp, R., et al. 2005. Multiple alignment of genomic sequences using CHAOS, DIALIGN and ABC. *Nucleic Acids Res.* 33, W532–W534.

Raphael, B., Zhi, D., Tang, H., et al. 2004. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Res.* 14, 2336–2346.

Schwartz, S., Kent, W.J., Smit, A., et al. 2003. Human-mouse alignments with BLASTZ. *Genome Res.* 13, 103–107.

Shapiro, L., and Stephens, A.B. 1991. Bootstrap percolation, the Schröder numbers, and the *N*-kings problem. *SIAM J. Discr. Math.* 4, 275–280.

Smith, T.F., and Waterman, M.S. 1981. Identification of common molecular subsequences. *Mol. Biol.* 147, 195–197.

Szklarczyk, R., and Heringa, J. 2006. AuberGenena sensitive genome alignment tool. *Bioinformatics* 22, 1431–1436.

Thompson, J.D., Higgins, D.G., and Gibson, T.J. 1994. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.* 22, 4673–4680.

Varre, J.-S., Delahaye, J.-P., and Rivals, E. 1999. Transformation distances: a family of dissimilarity measures based on movements of segments. *Bioinformatics* 15, 194–202.

Wallace, I., Sullivan, O., Higgins, D., et al. 2006. M-Coffee: combining multiple sequence alignment methods with T-Coffee. *Nucleic Acids Res.* 34, 1692–1699.

Weisstein, E.W. 2003. *CRC Concise Encyclopedia of Mathematics*, 2nd ed. CRC Press, Boca Raton, FL.

Yancopoulos, S., Attie, O., and Friedberg, R. 2005. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics* 21, 3340–3346.

Ye, L., and Huang, X. 2005. MAP2: multiple alignment of syntenic genomic sequences. *Nucleic Acids Res.* 33, 162–170.

Zhang, Y., and Waterman, M.S. 2005. An Eulerian path approach to local multiple alignment for DNA sequences. *Proc. Nat. Acad. Sci. USA* 102, 1285–1290.

Address correspondence to:
Dr. Lenwood S. Heath
Department of Computer Science
Virginia Tech
114 McBryde Hall
Blacksburg, VA 24061–0106

E-mail: heath@vt.edu