Random Shuffling Permutations of Nucleotides*

Shiquan Wu and Xun Gu
Center of Bioinformatics and Biological Statistics
Iowa State University, Ames, IA 50011, USA

Abstract

In this paper, we discuss a shuffling sequence problem: Given a DNA sequence, we generate a random sequence that preserves the frequencies of all mononucleotides, dinucleotides, trinucleotides or some high order base-compositions of the given sequence. Two quadratic running time algorithms, called Frequency-Counting algorithm and Decomposition-and-Reassemble algorithm, are presented for solving the problem. The first one is to count all frequencies of the mononucleotides, dinucleotides, trininucleotides, and any high order base-compositions in the given sequence. The second one is to generate a random DNA sequence that preserves the mononucleotides, dinucleotides, trinucleotides, or some high order base-compositions. The two algorithms are implemented into a program *ShuffleSeq* (in C) and is available at http://www.cs.iastate.edu/~ sqwu/ShuffleSeq.html.

Keywords Sequence, $r-$let, frequency, random, Decomposition-and-Reassemble algorithm.

1 Introduction

Markov Chain Monte Carlo (MCMC) method is extensively applied to gene finding and motifs identifying. Given a collection of known genes, we at first do training on the genes to get the initial and transition probabilities for nucleotides. Then based on these probabilities, we use Bayesian method to identify other unkown genes and the most significant patterns in any sequences based on posteriori probabilities (cf. [3, 5]). When apply MCMC method to finding genes, we at first generate a large number of sequences that preserve the initial and transition probabilities and then predict the locations of genes by expectations.

It is important to generate random sequences preserving the frequencies of all mononucleotides, dinucleotides, trinucleotides, and certain base-compositions (in high order Markov Chain Models) for finding genes in DNA sequences and identifying various motifs or other secondary and tertiary structures from the linear sequences of proteins (cf.[1, 4, 8]). Much work had been done for this problem on either algorithms and programs, or applications (cf.[1, 4, 8, 6, 7, 9]). However, algorithms for solving the problem are usually based on lower orders of base-compositions. For example, the algorithm designed by Kandel *et al* (cf.[8]) deals with lower order cases. The running time of the algorithm becomes exponential with respect to the orders of base-compositions when the orders are not fixed. More efficient algorithms are needed for solving the problem. On the other hand, fewer programs are available for generating such random sequences. In this paper, we discuss how to efficiently generate random sequences that preserve the frequencies of all mononucleotides, dinucleotides, trinucleotides or some high order base-compositions and design two quadratic running time algorithms and a program for the purpose.

Problem
Generally, the problem can be described as: Given a DNA sequence, we generate another random sequence that preserves the frequencies of all mononucleotides, dinucleotides, trinucleotides or any base-composition up to some orders.

For any sequence $S_1 = s_1 s_2 \cdots s_n$, a segment (or a block) $s_i s_{i+1} \cdots s_{i+k-1} (1 \leq i \leq n)$ is called a $k-$let of S_1 (cf.[1, 4]), or a base-composition of order k. For $k = 1, 2$, and 3, a $k-$let is a mononucleotide (singlet), dinucleotide (doublet), trinucleotide (triplet), respectively. Formally, we rephrase the problem in the following.

Shuffling Sequence Problem Let $S_1 = s_1 s_2 \cdots s_n$ be a DNA sequence and r an integer. Find a random DNA sequence S_2 such that for any $k \leq r$, any $k-$let has the same frequency in both S_1 and S_2.

When $r = 1$, S_2 is any permutation of all nucleotides $s_i (1 \leq i \leq n)$ of S_1. When $r = 2$, S_2 is

*Research supported in part by NIH Grant RO1 GM62118(to X.G) and NSF of China (19771025).

a dinucleotide permutation of S_1. The shuffling sequence problem is trivial for $r = 1$. Generally, for $r = 1, 2, \cdots$, when r is getting bigger and bigger, there are more and more (but then less and less) such S_2. When r is big enough, say $r = n$, there is a unique $S_2 = S_1$ (cf. [1, 4, 8]).

Previous work

Shuffling sequence problems were discussed by many researchers. Dinucleotide composition usage was discussed for sequence similarity based on permutation distances, and some methods were used to simulate random/nonrandom dinucleotide and codon usage (cf.[6, 7, 9]). Based on Euler tours in a directed graph, Altschul *et al* (cf.[1]) designed an algorithm to generate a random sequence with equal probability that preserves all mononucleotides, dinucleotides and codons. A swapping-based algorithm is given by Unger *et al* (cf.[13]). By "swapping operations" and Euler tours in a directed graph, Kandel *et al* presented a linear running time algorithm for solving a shuffling sequence problem that preserves all $r-$lets (cf.[8]). Coward designed a program for generating shuffling sequences that preserve all $k-$lets for each $k \leq r$ (cf.[4]). However, both Kandel's algorithm and Coward's program only deal with small $r-$lets. The running time becomes exponential with respect to r if r is not fixed and small.

This problem has great potential applications. Most studies directly or indirectly apply base-compositions in finding genes and identifying various motifs or other secondary and tertiary structures from the linear sequences (cf.[2, 4, 11]). Our purpose here is to design better algorithms and programs that improve Kandel's algorithm and Coward's program. These may be some useful tools for Monte Carlo methods in finding statistical significance for biological sequences.

Main results

In this paper, we design two quadratic running time algorithms and a program for finding solutions for the shuffling sequence problem.

(1) Frequency-Counting algorithm is to count the frequencies of all $r-$lets.

(2) Decomposition-and-Reassemble algorithm is to generate a random DNA sequence that preserves the frequencies of all $k-$lets for each $k \leq r$.

(3)The two algorithms are implemented into a program *ShuffleSeq* (in C).

Our algorithms are better than those designed by Altschul *et al* (cf.[1]), Kandel *et al* (cf.[8]), etc. The program is more general than Coward's (cf.[4]).

2 Algorithms

In this section, we design the two quadratic running time algorithms: Frequency-Counting algorithm and Decomposition-and-Reassemble algorithm. We at first obtain the principle of the algorithms by analysizing the structures of shuffling sequences and then design the algorithms based on the principle.

Principle of the algorithm

First of all, we analyze the structures that a new generated shuffling sequence must have so as to preserve all $k-$lets ($k \leq r$) for the given sequence S_1.

If $r = 1$, then any rearrangement of all s_i of S_1 can always preserve all singlets. If $r \geq 2$, we can not arbitrarily rearrange all s_i without changing the doublets. We must rearrange S_1 block by block. For example, for $r = 2$, by Kandel's "Swapping Algorithm" (cf.[8]), when we exchanges two blocks B and D in S_1, all singlets and doublets are still preserved (see Fig. 1(a) and (b)). Generally, all blocks can only be moved or swapped along S_1 under the restriction that all $k-$lets ($k \leq r$) must be preserved.

We now improve the "Swapping Algorithm". Denote $S_1 = ABCDE$, where A, B, C, D, E are blocks of S_1. Let b_1 and b_2 be $(r-1)-$lets. Each block (but A) starts with either b_1 or b_2 as shown in Fig. 1. At first, we take BC away from S_1. Because BC is followed by b_1 in D, we can then form a cycle with BC (i.e., BC is a cycle). The cycle contains b_2 (in C), we split the cycle at b_2 and get a new block CB. Finally, we insert CB between D and E. During the whole process, all $k-$lets ($k \leq r$) are preserved (i.e., while taking BC from S_1, A is still followed by b_1 in D. While forming the cycle, C is connected to b_1 in B. While splitting the cycle and inserting CB, D is followed by b_2 in CB and CB is connected to b_2 in E. Therefore all $k-$lets are preserved in S_2) (see Fig. 1(c) and (b)). Fig. 1(d) and (e) show that $\underline{S_2} = ACBD$ is obtained from $\underline{S_1} = ABCD$ by the process in Fig. 1(c), but not "Swapping Algorithm". Moreover, in order to preserve all $k-$lets ($k \leq r$), each b_i must be an $(r-1)-$let. Otherwise, if some b_i is a $k-$let ($k \leq r-2$), then some $r-$lets will be broken or created.

We can construct a series of cycles from S_1 and insert them back to S_1 one after another to form a new S_2. We call this a cycle Decomposition-and-Reassemble process. It has an advantage over the "Swapping Algorithm" (cf.[8]). The "Swapping Algorithm" each time choose four $(r-1)-$lets and gives rise to an $O(n^4)$ running time of the algorithm. We scan S_1 for two $(r-1)-$lets in our cycle Decomposition-and-Reassemble process and can design a quadratic

algorithm. Therefore, we take this process as the principle of our algorithm.

Principle of Decomposition-and-Reassemble If we take a cycle from S_1, split the cycle at some $(r-1)-$let b, and insert it at some $(r-1)-$let b in S_1, then all $k-$lets $(k \leq r)$ are preserved (Fig. 1(c)).

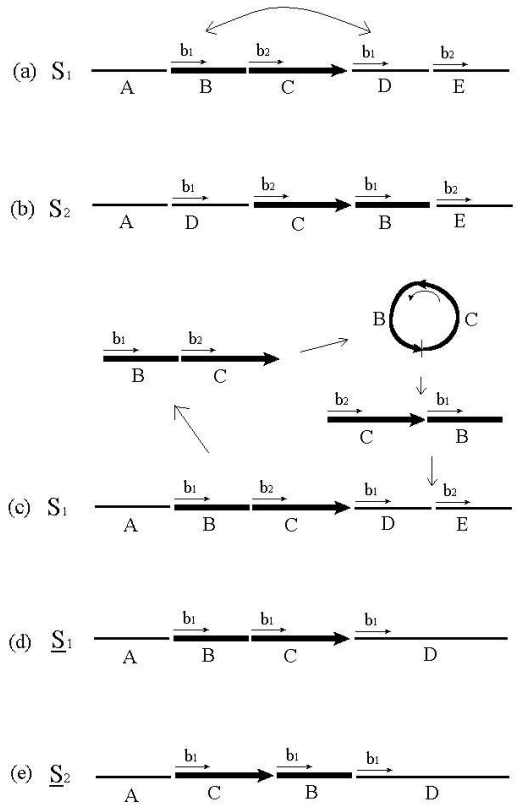

Figure 1: Structure analysis: b_1 and b_2 are $(r-1)-$lets. (a)Kandel's "Swapping Algorithm" exchanges Block B and D in S_1. (b)$S_2 = ADCBE$ is obtained from $S_1 = ABCDE$ by (a) or (c). (c)The process of cycle Decomposition-and-Reassemble: At first, BC is taken from S_1 to form a cycle. Next, the cycle is split into a new block CB. And then, CB is inserted between D and E. (d)The process in (c) can be applied to $ACBD$, which contains three copies of an $(r-1)-$let b_1. (e)$\underline{S}_2 = ACBD$ is obtained from $\underline{S}_1 = ABCD$ by the process in (c), but (a).

Frequency counting algorithm

This is an obvious algorithm. It loops over all positions $s_i (1 \leq i \leq n)$. At each position s_i, the algorithm check all possible $k-$lets: $R = s_i s_{i+1} \cdots s_{i+k-1} (k = 1, 2, \cdots, r)$. Compare R with all $k-$lets $R_1, R_2, \cdots, R_p$ that had been previously found, if $R = R_j$, then augment the frequency of R_j by one. Otherwise, R is a new $k-$let, i.e., $R \neq R_j$ for any j. Denote $R_{p+1} = R$ and define that its frequency is equal to one at this moment. When the algorithm terminates, the frequencies of all $k-$lets $(k \leq r)$ are obtained.

Algorithm	**Frequency-Counting**
Input	DNA sequence: $S = s_1 s_2 \cdots s_n$.
Output	Frequencies of all $k-$lets $(k \leq r)$.
Initial:	$p = 1$; $\mathcal{R} = \emptyset$ (all $k-$lets).
	$\mathcal{F} = \emptyset$ (all frequencies).
Loop:	For $i = 1, 2 \cdots, n$.
	For $k = 1, 2 \cdots, r$.
	Compare each $R = s_i s_{i+1} \cdots s_{i+k-1}$
	with all $R_j \in \mathcal{R} (j = 1, 2, \cdots, p)$.
	If $R = R_j$, then $f_j + +$.
	If $R \neq R_j$ for all $R_j \in \mathcal{R}$, then
	$p + +$, define $R_p = R$, $f_p = 1$,
	add them to $\mathcal{R}$, $\mathcal{F}$,respectively.

Theorem 1 *The running time of the Frequency-Counting algorithm is* $O(n^2 r^2)$.

Proof There are $O(n^2 r)$ pairs of $k-$lets. We can know whether each pair is the same by comparing at most r positions of them. This follows the running time $O(n^2 r^2)$.

Decomposition-and-Reassemble algorithm

The Decomposition-and-Reassemble algorithm is to generate a random DNA sequence that preserves the frequencies of all $k-$lets of the given DNA sequence $(k \leq r)$. It is based on the principle of Decomposition-and-Reassemble and consists of two steps: Decomposition step and reassemble step. In the decomposition step, we decompose the given DNA sequence into a series of cycles and a path. Each cycle shares some common $(r-1)-$let with the path or some other cycles. In the reassemble step, we reassemble all the cycles into a DNA sequence. If we reassemble each cycle in the exact reverse order as we get in the decomposition step, then we can get the original DNA sequence. However, for many cycles, they can be put back to either their original places, or other places with the same/similar structures in the sequence. In this way, many new DNA sequences can be generated. The details of the steps are described in the following.

Decomposition Step Let $S = s_1 s_2 \cdots s_n$. Denote $P_0 = S_1$ as the initial path. We now decompose it into a cycle and a path. Start from s_1, for $i = 2, 3, \cdots, n$, compare s_i with all s_j $(j < i)$. If $s_{i_1} s_{i_1+1} \cdots s_{i_1+r-2} = s_i s_{i+1} \cdots s_{i+r-2}$. Then $s_{i_1} s_{i_1+1} \cdots s_i$ is a cycle. The

path

$$P_0 = s_1 s_2 \cdots s_{i_1-1} s_{i_1} s_{i_1+1} \cdots s_i s_{i+1} \cdots s_n \qquad (1)$$

is then decomposed into a cycle C_1 and a new path P_1 (see the first step in Fig. 2):

$$\begin{cases} C_1 = s_{i_1} s_{i_1+1} \cdots s_i, \\ P_1 = s_1 s_2 \cdots s_{i_1-1} s_i s_{i+1} \cdots s_n. \end{cases} \qquad (2)$$

C_1 and P_1 share a common part $D_1 = s_{i_1} s_{i_1+1} \cdots s_{i_1+c_1}$. To preserve all $r-$lets, we must have $c_1 \geq r-2$. We call this part the support of C_1 on P_1.

Decompose P_1 again, we can get another cycle C_2 and another new path P_2 with a support D_2 of C_2 on P_2. Repeat the decomposition preccess again and again, we get a series of cycles C_i and paths P_i together with supports $D_i (i = 1, 2, \cdots, m)$. We can also express the decomposition by some equations similar to Eq. (1) and (2). It is possible that some support D_p will be decomposed into other cycle $C_q (q > p)$ later. P_m contains an $(r-1)-$let common to some cycles. The whole decomposition process is shown in Fig.2 for a short DNA sequence.

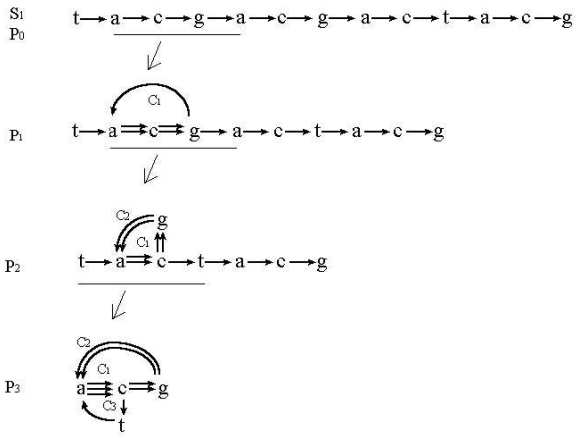

Figure 2: The decomposition process for $S_1 = tacgacgactacg$: $(1) P_0 (= S_1)$ is decomposed into $C_1 = acga$ and $P_1 = tacgactacg$ with $D_1 = \{a, c, g\}$. $(2) P_1$ is decomposed into $C_2 (= C_1)$ and $P_2 = tactacg$ with $D_2 = \{a, c\}$. $(3) P_3$ is decomposed into $C_3 = tact$ and the final path $P_3 = acg$. $D_3 = \{a, c\}$.

Reassemble Step The reassemble step pursues an inverse process to the decomposition step. We begin with the final path P_m and insert all cycles back to the path one after another to get a new DNA sequence. If we choose each cycle C_i in the exact reverse order and put them into the same places as they were generated in the decomposition step, we then get the original DNA sequence S_1. However, we can choose different orders and various inserting positions while inserting these cycles. Since we are required to preserve all $k-$lets $(k \leq r)$, the number of possible ways depends on r. If r is small, then there are more ways for inserting the cycles. For example, for $r = 1$, we can insert all cycles arbitrarily. If r is bigger, then fewer options we can have. We have the following cases:

Case 1 $r = 1$. In this case, we can insert all cycles arbitrarily and get a sequence that preserves the same numbers of all nucleotides. It is equivalent to rearranging all s_i for the given S_1.

Case 2 $r = 2$. This is a simple case, in order to preserve all mononucleotides and dinucleotides, we can insert a cycle C into a path P if and only if C and P share a common nucleotide, i.e., a $1-$let. We split both C and P at the common nucleotide and then join them together as shown in Fig.3.

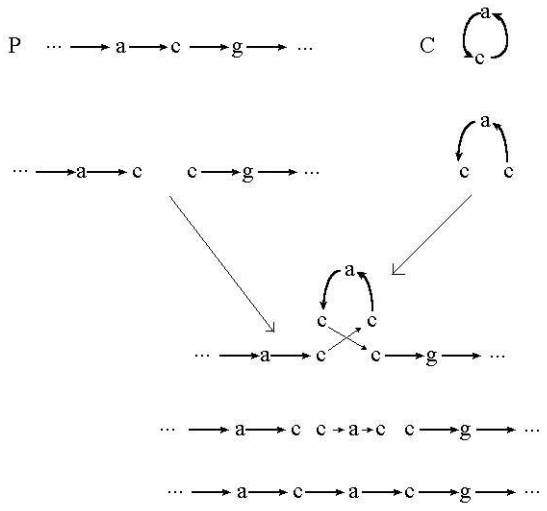

Figure 3: Reassemble Step: Insert C into P preserving all $1-, 2-$lets. $(1) P$ and C share a common nucleotide "c". (2)Split P and C at "c". (3)Join P and C at "c".

Case 3 $r = 3$. In order to preserve all mononucleotides, dinucleotides, and trinucleotides, we can insert a cycle C into a path P if and only if C and P share a common dinucleotide, i.e., a $2-$let. Suppose C and P contains a dinucleotide $s_i s_{i+1}$ in common.

$$\begin{cases} P = s_1 s_2 \cdots s_{i-1} \underline{s_i s_{i+1}} s_{i+2} \cdots s_n, \\ C = c_p c_{p+1} \cdots c_{i-1} \underline{s_i s_{i+1}} c_{i+2} \cdots c_q \end{cases} \qquad (3)$$

We at first split both C and P at s_i (or s_{i+1}):

$$\begin{cases} P_1 = s_1 s_2 \cdots s_{i-1} \underline{s_i} \\ P_2 = \underline{s_i} s_{i+1} s_{i+2} \cdots s_n, \\ C_1 = c_p c_{p+1} \cdots c_{i-1} \underline{s_i} \\ C_2 = \underline{s_i} s_{i+1} c_{i+2} \cdots c_q. \end{cases} \quad (4)$$

And then join them together at s_i, which goes this way: $P_1 - C_2 - reverse(C_1) - P_2$. We have

$$\begin{aligned} P = \quad & s_1 s_2 \cdots s_{i-1} \underline{s_i s_{i+1}} c_{i+2} \cdots c_q \\ & c_{p-1} \cdots \ c_{i-1} \ \underline{s_i s_{i+1}} s_{i+2} \cdots s_n. \end{aligned} \quad (5)$$

The process is shown in Fig.4.

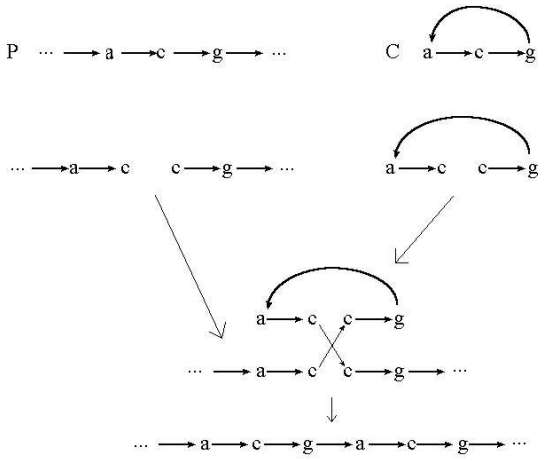

Figure 4: Reassemble Step ($r = 3$): Insert C into P preserving all $r-$lets ($r \leq 3$). (1)P and C share a common 2−let "cg". (2)Split P and C at c. (3)Join P and C at c.

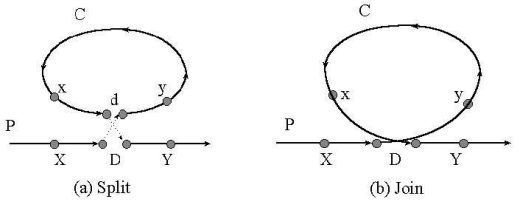

Figure 5: Reassemble Step (general r): (a) C and P share a common $(r-1)-$lets: $[x, y] = [X, Y]$. Split C and P at D. (b) Join P and C at D. $[x+, y+]$ is preserved as $[x+, Y+]$. $[x-, y-]$ is preserved as $[x-, Y-]$. $[X+, Y+]$ is preserved as $[X+, y+]$. $[X-, Y-]$ is preserved as $[X-, y-]$.

Case 4 General case: $r \leq n$ This case is similar to that for $r = 3$. In order to preserve all $r-$lets, we can insert a cycle C into a path P if and only if both C and P share a common $(r-1)-$let. The process of splitting and joining is the same as that for $r = 3$ and is shown in Fig. 5, where we denote $X+$ (or $X-$) a position on the right (or left) hand side of X. Then $[X+, Y-]$ stands for a segment from some position on the right of X to some position on the left of Y.

We now design the algorithm based on our previous discussions. The algorithm consists of decomposition step and reassemble step.

Algorithm **Decomposition-and-Reassemble**
Input DNA sequence: $S = s_1 s_2 \cdots s_n$.
Output New DNA sequence preserving all frequencies of all $k-$lets ($k \leq r$).

Decomposition Step:
$p = 1$. $P = s_1 s_2 \cdots s_i s_{i+1} \cdots s_n$.
For $i = 1, 2 \cdots, n$. $j = 1, 2 \cdots, i-1$.
If $s_j = s_i, \cdots, s_{j+r-1} = s_{i+r-1}$, then
define a cycle $C_p = s_j s_{j+1} \cdots s_i$. $p++$;
Update $P = s_1 s_2 \cdots s_{j-1} s_i s_{i+1} \cdots s_n$.
Repeat the process until convergence.

Reassemble Step:
Denote P the final path and $C_p (1 \leq p \leq m)$ all cycles obtained in the decomposition step.
For $i = 1, 2 \cdots, m$, randomly choose a cycle C_i that shares a common $(r-1)-$let with P.
Split and join C and P as shown in Eq. (3,4,5) and Fig. (3,4,5).

Theorem 2 *The running time of Decomposition-and-Reassemble algorithm is* $O(n^2 r^2)$.

Proof By Theorem 1, the decomposition step and the reassemble step.

Our two algorithms have a running time $O(n^2 r^2)$. However, r can not be too large in practical applications. Therefore we can assume that r is a constant (e.g., $r \leq 1000$). The two algorithms then have a running time $O(n^2)$.

Our Decomposition-and-Reassemble algorithm is better than previous algorithms, such as the Euler-tour-based algorithm and the swapping-based algorithm designed by of Kandel *et al* (cf.[8]).

If we choose each cyccle with an equal probability and insert it into all possible positions with an equal probability, then each possible valid random sequence is uniformly generated.

3 Experimental examples

The two algorithms, Frequency-Counting algorithm and Decomposition-and-Reassemble algorithm,

are implemented into a program *ShuffleSeq* in C and is available at http://www.cs.iastate.edu/~sqwu/ShuffleSeq.html.

The program takes a DNA sequence in FASTA format as the input, then finds the frequencies of all $k-$lets ($k \leq r$). Finally it generates a random sequence that preserves the frequencies of all $k-$lets ($k \leq r$). The program deals with $r-$lets for any $r \leq 1000$ and is more general than Coward's (cf.[4]). We now use the program to some DNA sequences.

Example 1 For $S_1 = acgaccgacctta$ and $r = 2$. The program each time uniformly generates one of its 18 dinucleotide permutations.

acccgacgactta	acccgacttacga	acccttacgacga
accgaccgactta	accgaccttacga	accgacgacctta
accgacttaccga	accttaccgacga	accttacgaccga
acgacccgactta	acgacccttacga	acgaccgacctta
acgaccttaccga	acgacgaccctta	acgacttacccga
acttacccgacga	acttaccgaccga	acttacgacccga

Example 2 Choose a gene from E.Coli K12:

$S_1 =$ *tcagtttctgtacccgcgtgattggagtaaatga*
tgcagttctcgaaaatgcatggccattggcaa

For $r = 6$, by the program, we obtain

$S_2 =$ *tcagtttctgtacccgcgtgattggagtaaat*
gatgcagttctcgaaaatgcatggccattggcaa.

They have the same $r-$lets and frequencies ($r \leq 6$).

4 Discussion

We discuss the shuffling sequence problem for DNA sequences. The algorithms and program are valid for other biological sequences. The shuffling sequence problem has great potential applications in gene finding and motif identifying. Whenever the MCMC method is applied, our algorithms can be used to generate a collection of random sequences for finding statistical significance, especially for predicting genes and motifs, or any kinds of genomic data processing.

For example, coiled-coil-like motifs are some kinds of segments that highly repeat some patterns in linear sequences. Some programs were designed to detect the coiled-coil-like motifs for viral membrane-fusion proteins (cf.[10, 12]). Our algorithms and program can be applied to this topic for predicting the locations that coiled-coil-like motifs mostly possible occur.

It is interesting to search for genes by using base-composition statistics. Our algorithms and program can be some useful tools in this area.

References

[1] Stephen F. Altschul and Bruce W. Ericksont, *Significance of Nucleotide Sequence Alignments: A Method for Random Sequence Permutation That Preserves Dinucleotide and Codon Usage*, Mol. Biol. Evol. 2(6): 526-538.1985.

[2] Pierre Baldi and Pierre-Francois Baisnee, *Sequence analysis by additive scales: DNA structure for sequences and repeats of all lengths*, Bioinformatics 16: 865-889. 2000.

[3] Borodovsky M. and McIninch J. *GeneMark: parallel gene recognition for both DNA strands*, Computers and Chemistry, 17:123-133. 1993

[4] Eivind Coward, *Shufflet: shuffling sequences while conserving the k-let counts*, Bioinformatics 15: 1058-1059.1999.

[5] Eddy, S. R, *Profile hidden Markov models*, Bioinformatics 14(9):755-63, 1998.

[6] Fitch, W. M.,*Calculating the expected frequencies of potential secondary structure in nucleic acids as a function of stem length, loop size, base composition and nearest-neighbor frequencies*, Nucleic Acids Res. 11:4655-4663. 1983.

[7] Fitch, W. M.,*Random sequences* J. Mol. Biol. 163:171-l 76. 1983.

[8] Kandel, D. , Y. Matias, R. Unger, P. Winkler, *Shuffling Biological Sequences*, Discrete Applied Mathematics, 71: 171-185, 1996.

[9] Lipman, D. J., W. J. Wilbur, T. F. Smith, and M. S. Waterman, *On the statistical significance of nucleic acid similarities*, Nucleic Acids Res. 12:2 15-226.1984.

[10] Malashkevich, V. N., M. Singh, and P. S. Kim *The trimer-of-hairpins motif in membrane fusion: Visna virus*, PNAS, 98: 8502-8506. 2001.

[11] Peter Schattner, *Searching for RNA genes using base-composition statistics*, Nucleic Acids Research, 30(9).2002.

[12] Singh, M., B. Berger, and P. S. Kim *LearnCoil-VMF: Computational Evidence for Coiled-coil-like Motifs in Many Viral Membrane-fusion Proteins*, J. Mol. Biol. 290:1031-1041. 1999.

[13] Unger, R., G. Avrahami, D. Harel, and J. L. Sussman, *Simple general shuffling scheme which preserves fragment frequencies up to any required length.* In *Proc. Macromolecules, Genes, and Computers Conference*, 1986.