A linear-time algorithm that avoids inverses and computes jackknife (leave-one-out) products like convolutions or other operators in commutative semigroups

John Spouge (spouge@nih.gov)
National Center for Biotechnology Information https://orcid.org/0000-0001-6207-1419
Joseph M. Ziegelbauer
National Institutes of Health
Mileidy Gonzalez
Lenovo HPC and AI

Research

Keywords: commutative semigroup, leave-one-out, jackknife products, segment tree, data structure

Posted Date: March 26th, 2020

DOI: https://doi.org/10.21203/rs.3.rs-18420/v1

Version of Record: A version of this preprint was published on September 19th, 2020. See the published version at https://doi.org/10.1186/s13015-020-00178-x.

Abstract

Background: Data about herpesvirus microRNA motifs on human circular RNAs suggested the following statistical question. Consider independent random counts, not necessarily identically distributed. Conditioned on the sum, decide whether one of the counts is unusually large. Exact computation of the p-value leads to a specific algorithmic problem. Given n elements $g_0,g_1,...g_{n-1}$ in a set with the closure and associative properties and a commutative product without inverses, compute the jackknife (leave-one-out) products gbar;=$g_0,g_1,...g_{j-1}\,g_{j+1}...g_{n-1}$ (0≤$j<n$).

Results: This article gives a linear-time Jackknife Product algorithm. Its upward phase constructs a standard segment tree for computing segment products like $g_{[i,j)}=g_i g_{i+1}...g_{j-1}$; its novel downward phase mirrors the upward phase while exploiting the symmetry of and its complement gbar;$_j$. The algorithm requires storage for elements of and only about products. In contrast, the standard segment tree algorithms require about n products for construction and $\log_2 n$ products for calculating each gbar;$_j$, i.e., about products $n \log n$ in total; and a naïve quadratic algorithm using n-2 element-by-element products to compute each gbar;$_j$ requires n (n-2) products.

Conclusions: In the herpesvirus application, the Jackknife Product algorithm required 15 minutes; standard segment tree algorithms would have taken an estimated 3 hours; and the quadratic algorithm, an estimated 1 month. The Jackknife Product algorithm has many possible uses in bioinformatics and statistics.

Background

A Biological Question: Circular RNAs (circRNAs) are single-stranded noncoding RNAs that can inhibit another RNA molecule by binding to it, mopping it up like a sponge. During herpesvirus infection, human hosts produce circRNAs with target sites that may bind herpesvirus microRNA (miRNA) [1]. (See Fig. 1.) Given a sequence motif, e.g., a target site for a miRNA, researchers counted how many times the motif occurs in each circRNA sequence. They then posed a question: is the motif unusually enriched in any of the circRNAs, i.e., does any circRNA have too many occurrences of the motif to be explained by chance alone? If “yes”, the researchers could then focus their further experimental efforts on those circRNAs.

A Statistical Answer

Figure 1 illustrates a set of circRNAs with varying length, with a single miRNA motif occurring as indicated on each circRNA. Let $i = 0,1,...,n-1$ index the circRNAs; the random variate X_i count the motif occurrences in the i-th circRNA; $K(i)$ equal the observed count for X_i; and the sum $S = \sum_{i=0}^{n-1} X_i$ count the total motif occurrences among the circRNAs, with observed total $K = \sum_{i=0}^{n-1} k(i)$.

The following set-up provides a general statistical test for deciding the biological question. Let $\{X_i : i = 0,1,\ldots,n-1\}$ represent independent random counts (i.e., non-negative integer random variates), not necessarily identically distributed, with sum $S = \sum_{i=0}^{n-1} X_i$. Given observed values $\{X_i = k(i) : i = 0,1,\ldots,n-1\}$ with observed sum $K = \sum_{i=0}^{n-1} k(i)$, consider the computation of the conditional p-values $\mathrm{P}\{X_i \geq k(i) | S = K\}$ $(i = 0,1,\ldots,n-1)$. The conditional p-values can decide the question: "Is any term in the sum unusually large relative to the others?"

The abstract question in the previous paragraph generalizes some common tests. For example, the standard 2 × 2 Fisher exact test [2, p. 96] answers the question in the special case of *n = 2 categories: each X_i has a binomial distribution with common success probability P, conditional on known numbers of trials N_i (i=0,1). Although the Fisher exact test generalizes directly to a single exact p-value for a 2×n table [3], the generalization can require prohibitive amounts of computation. The abstract question corresponds to a computationally cheaper alternative that also decides which columns in the 2×n table are unusual [4].*

To derive an expression for the conditional p-value, therefore, let $g_i[k] = \mathrm{P}\{X_i = k\}$ be given, so the array $g_i = (g_i[0], g_i[1], \ldots, g_i[K])$ gives the distribution of X_i, truncated at the observed total $K = \sum_{i=0}^{n-1} k(i)$. Because g_i is a truncated probability distribution, $g_i \varepsilon$ G, the set of all real (K+1)-tuples such that $g[k] \geq 0 \quad (k = 0,1,\ldots,K)$ and $\sum_{k=0}^{K} g[k] \leq 1$. To calculate the distribution of the sum $S = \sum_{i=0}^{n-1} X_i$ for S ≤ K, define the truncated convolution operation $g = g' \circ g''$, for which $g[k] = \sum_{j=0}^{k} g'[j] g''[k-j]$ (k = 0,1,…,K). Hereafter, the operation "∘" is often left implicit: $g' \circ g'' = g'g''$.

Let $\bar{g} = g_0 g_1 \cdots g_{n-1}$, so $g[k] = \mathrm{P}\{S = k\}$ $(k = 0,1,\ldots,K)$. Define the "jackknife products" $\bar{g}_j = g_0 g_1 \cdots g_{j-1} g_{j+1} \cdots g_{n-1}$ $(0 \leq j < n)$ (implicitly including the products $\bar{g}_0 = g_1 g_2 \cdots g_{n-1}$ and $\bar{g}_{n-1} = g_0 g_1 \cdots g_{n-2}$). The jackknife products contain the same products as $\bar{g}$, except that in turn each skips over g_j (0 ≤ j < n). Like the jackknife procedure in statistics, therefore, jackknife products successively omit each datum in a dataset [5].

With the jackknife products in hand, the conditional p-values are a straightforward computation:

$$\mathrm{P}\left\{X_i \geq k(i) \middle| S = K\right\} = \frac{\sum_{k=k(i)}^{K} g_i[k]\,\bar{g}_i[K-k]}{\bar{g}[K]}.$$

1

.

With respect to Eq. (1) and the biological question in Fig. 1, Appendix B gives the count $\bar{g}$ *[K] of the ways that n circular lattices of known but varying length (circRNAs) may contain K lattice segments of equal length (miRNA motifs), the count* $\bar{g}_i$ *[K-k] of the ways that the i-th circle may contain k segments, and the count* $\bar{g}_i$*[K-k] of the ways that all circles but the i-th may contain k segments. For combinatorial probabilities like* $\mathrm{P}\left\{X_i \geq k(i) \middle| S = K\right\}$*, Eq. (1) remains relevant, even if {*g_i*[k]} are counts instead of probabilities. The biological question therefore exemplifies a commonplace computational need in applied combinatorial probability.*

The Discussion indicates some obstacles in computing Eq. (1) with transforms, so this article pursues direct exact calculation of $\mathrm{P}\left\{X_i \geq k(i) \middle| S = K\right\}$*. The product forms of* $\bar{g}$ *and {*$\bar{g}_j$*} suggest that any efficient algorithm may be abstracted to broaden its applications, as follows.*

Semigroups, Groups, and Commutative Groups

Let $(G, \circ)$ *denote a set G with a binary product* $g \circ g'$ *on its elements. Consider the following properties [6].*

(1)

Closure: $g \circ g' \in G$ for every $g, g' \in G$

(2)

Associative: $(g \circ g') \circ g'' = g \circ (g' \circ g'')$ for every $g, g', g'' \in G$

(3)

Identity: There exists an identity element $e \in G$*, such that* $e \circ g = g \circ e = g$ *for every g ∈ G*

(4)

Commutative: $g \circ g' = g' \circ g$ for every $g, g' \in G$

If the Closure and Associative properties hold, $(G, \circ)$ *is a semigroup. Without loss of generality, we assume below that the Identity property holds. If not, adjoin an element e∈ G, such that* $e \circ g = g \circ e = g$

for every g∈ G. In addition, if the Commutative property holds for every g, g'∈ G, the semigroup $(G, \circ)$ *is commutative. Unless stated otherwise hereafter,* $(G, \circ)$ *denotes a commutative semigroup. The Jackknife Product algorithm central to this article is correct in a commutative semigroup.*

(5)

Inverse: For every g ∈ G, there exists an inverse $g^{-1} \in G$*, such that* $g \circ g^{-1} = g^{-1} \circ g = e$

As shown later, the Jackknife Product algorithm does not require the Inverse property. In passing, note that the convolution semigroup relevant to the circRNA-miRNA application lacks the Inverse property. To demonstrate, let X, Y ≥ 0 be independent integer variates. The identity e for convolution corresponds to the variate Z = 0, because 0 + X = X + 0 = X for every variate X. If X = Y = 0, however, the independence of X and Y implies that both are constant and therefore X = Y = 0. In the relevant convolution semigroup, therefore, all elements except the identity e lack an inverse.

The non-zero real numbers under ordinary multiplication form a commutative semigroup $(G, \circ)$ *with the Inverse property. They provide a familiar setting for discussing some algorithmic issues when computing* $\{\bar{g}_j\}$. Let $\bar{g} = g_0 g_1 \cdots g_{n-1}$ *be the usual product of n real numbers, and consider the toy problem of computing all jackknife products* $\{\bar{g}_j\}$ *that omit a single factor* g_j *(0 ≤ j < n) from* $\bar{g}$*. Inverses* $\{g_j^{-1}\}$ *are available, so an obvious algorithm computes* $\bar{g}$ *and then* $\{\bar{g}_j = \bar{g} g_j^{-1}\}$ *with n inverses and 2n-1 = (n-1)+n products. If the inverses were unavailable, however, the naïve algorithm using n-2 element-by-element products to compute each* $\{\bar{g}_j\}$ *would require n(n-2) products. The quadratic time renders the naïve algorithm impractical for many applications.*

The Theory section (next) notes that a standard data structure called a segment tree can calculate the jackknife products $\{\bar{g}_j\}$ *in time O (nlogn), fast enough for many applications. For a commutative semigroup* $(G, \circ)$*, however, a Jackknife Product algorithm can avoid inverses and still manage to reduce the computational time from O (nlogn) to O n. The Jackknife Product algorithm has three phases. Its upward phase just constructs a segment tree; its downward phase exploits the symmetry of* g_j *and its complement* $\bar{g}_j$ *to mirror the upward phase while computing* $\{\bar{g}_j\}$*; and its final transposition phase then swaps successive pairs in an array. With in-place computations requiring only the space for the segment tree, the Jackknife Product algorithm avoids inverses yet still requires only about 3n products and storage for 3n numbers. It is therefore surprisingly economical, even when compared to the obvious algorithm*

using inverses. Because convolutions in the statistical setting form a commutative semigroup $(G, \circ)$ *without inverses, the Jackknife Product algorithm already has broad applicability.*

Theory

Appendix A proves the correctness of the Jackknife Product algorithm given below.

The Jackknife Product Algorithm: Let $(G, \circ)$ *be a commutative semigroup. The Jackknife Product algorithm has three phases: upward, downward, and transposition. The upward phase simply constructs a segment tree, a standard data structure that Fig. 2 illustrates without the tree root. A segment tree can efficiently compute segment products* $g_{[i,j)} = g_i g_{i+1} \cdots g_{j-1}$*, from which jackknife products like* $\left\{ \bar{g}_j = g_{[0,j)} g_{[j+1,n)} \right\}$ *rapidly follow. The three phases yield a simpler algorithm if n = n$^{\Phi}$ = 2^m is a binary power. To recover the simpler algorithm from them, pad {g$_j$} on the right with copies of the identity e up to n$^{\Phi}$ elements, where n$^{\Phi}$ = 2^m is the smallest binary power greater than or equal to n, i.e., replace {g$_j$} with* $\{g_0, g_1, \ldots, g_{n-1}, e, \ldots, e\}$*, with n$^{\Phi}$ - n copies of e. The simpler algorithm can therefore restrict its input to n$^{\Phi}$ = 2^m elements without loss of generality. The algorithms given below therefore sacrifice some simplicity, but they may save almost a factor of 2 in storage and time by omitting the padded copies of e. In any case, the simpler algorithm can always be recovered from the phases for general n given here, if desired.*

We start with notational preliminaries. Define the floor function [x] = max {j : j≤x} and the ceiling function [x] = max {j : x≤j} (both standard); and the binary right-shift function ρ (j) = [j/2]. Other quantities also smooth our presentation. Given a product $\bar{g} = g_0 g_1 \cdots g_{n-1}$ *of interest, define* $m = \lceil \log_2 n \rceil$ and $n_k = \lceil n 2^{-k} \rceil$ for $0 \le k < m$*. Below, the symbol "□" connotes the end of a proof.*

The Upward Phase

The upward phase starts with the initial array $L_0[j] = g_j \ (0 \le j < n)$ *and simply computes a standard (but rootless) segment tree consisting of segment products L$_k$[j] for* $j = 0, 1, \ldots, n_k - 1$ and $k = 0, 1, \ldots, m-1$*.*

The upward phase

for $(j=0;\ j<n;\ j++)$ do

$L_0[j]=g_j;$

for $(k=1;\ n_k>2;\ k++)$ do

for $(j=0;\ j<\rho(n_{k-1});\ j++)$ do

$L_k[j]=L_{k-1}[2j]\circ L_{k-1}[2j+1];$

if (n_{k-1} is odd) $L_k[\rho(n_{k-1})]=L_{k-1}[n_{k-1}-1];$

Comments

(1) If $n = n^{\Phi} = 2^m$ is a binary power, $\rho(n_{k-1})=n_k=2^{m-k}$ and the final line in the upward phase can be omitted. (2) Of some peripheral interest, Laaksonen [7] gives the algorithm in a different context, embedding a binary tree in a single array of length O(n). If any $L_0[j] = g_j$ changes, he also shows how to update the single array with O(log n) multiplications. If the downward phase (next) does not overwrite the segment tree $\{L_k\}$ by using in-place computation, it permits a similar update.

The Downward Phase

The transposition function $\tau(j)=j+(-1)^j$ transposes adjacent indices, e.g., $(L_{\tau(0)},L_{\tau(1)},L_{\tau(2)},L_{\tau(3)})=(L_1,L_0,L_3,L_2))$. We also require $\alpha_k(j)=\min\{\tau\rho(j),n_k-1\}$ for $0\le j<n_{k-1}$ and $1\le k<m$, the index of "aunts", as illustrated by Fig. 3. Just as Fig. 2 illustrates a rootless segment tree in the upward phase, Fig. 3 illustrates the corresponding rootless complementary segment tree in the downward phase.

The downward phase computes complementary segment products $\bar{L}_k[j]$ for $j=0,1,\ldots,n_k-1$ and $k=m-1,m-2,\ldots,0$.

The downward phase

$\bar{L}_{m-1}[0] = L_{m-1}[0]$;

$\bar{L}_{m-1}[1] = L_{m-1}[1]$;

---

for $(k = m-1;\ k > 0;\ k--)$ do

for $(j = 0;\ j < 2\rho(n_{k-1});\ j++)$ do

$\bar{L}_{k-1}[j] = L_{k-1}[j] \circ \bar{L}_k[\alpha_k(j)]$;

if (n_{k-1} is odd) $\bar{L}_{k-1}[n_{k-1}-1] = \bar{L}_k[\alpha_k(n_{k-1}-1)]$;

Comments

(1) If n = $n^{\Phi} = 2^m$ is a binary power, $\rho(n_{k-1}) = n_k = 2^{m-k},\ \alpha_k(j) = \tau\rho(j)$, and the final line in the downward phase can be omitted. (2) The downward phase can be modified in the obvious fashion to permit in-place calculation of $\bar{L}_{k-1}[j]$ from $L_{k-1}[j]$, reducing total memory allocation by about 2.

As Appendix A proves, the final array $\bar{L}_0$ has elements $\bar{L}_0[\tau(j)] = \bar{g}_j\ (0 \le j < 2\rho(n))$, with an additional final element $\bar{L}_0[n-1] = \bar{g}_{n-1}$ if $n_0 = n$ is odd, so the Jackknife Product algorithm ends with a straightforward transposition phase.

Transposition phase

for $(j = 0;\ j < 2\rho(n);\ j++)$ do

$\bar{g}_j = \bar{L}_0[\tau(j)]$;

if (n is odd) $\bar{g}_{n-1} = \bar{L}_0[n-1]$;

Comments

The transposition phase can permit an in-place calculation of $\{\bar{g}_j\}$ to overwrite $\bar{L}_0$.

Computational Time and Storage

Note $n_j = \lceil n2^{-j} \rceil$, so $0 \le n_j - n2^{-j} < 1$. *To compute* L_k *from* L_{k-1} *or to compute* $\bar{L}_k$ *from* L_k *and* $\bar{L}_{k+1}$, *the Jackknife Product algorithm requires* n_k *products. For large n , therefore, the upward phase computing the segment tree requires about* $\sum_{j=1}^{m-1} n_j \approx \sum_{j=1}^{\infty} n2^{-j} = n$ *products; the downward phase, about* $\sum_{j=0}^{m-2} n_j \approx 2n$ *products. Likewise, if the downward and transposition phases compute in place by replacing* L_k *with* $\bar{L}_k$ *and* $\bar{L}_0$ *with* $\{\bar{g}_j\}$, *the algorithm storage is* $\sum_{j=0}^{m-1} n_j \approx 2n$ *semigroup elements. Each of the three estimates just given for products and storage have an error bounded by >m =[*$\log_2$ *n]. Although the case of general n could be handled by the algorithm for binary powers* $n^{\Phi} = 2^m$ *by setting m =[*$\log_2$ *n] and* $g_n = g_{n+1} = \ldots = g_{n^*-1} = e$, *the truncated arrays in the Jackknife Product algorithm for general n save about a factor of 1 ≤ n*/n<2 in both products and storage.*
As written, the conditional copy statements at the end of the upward and downward phases replicate elements already in storage. If the downward phase of the Jackknife Product algorithm is implemented with in-place computation of $\bar{L}_{k-1}[j]$ from $L_{k-1}[j]$, *the copy statements ensure that the algorithm never overwrites any array element it needs later. Some statements may copy some elements more than once (and therefore unnecessarily), but a negligible m =[*$\log_2$ *n] copies at most are unnecessary.*

The complementary segment tree in Fig. 3 implicitly indicates the nodes in the segment tree required to compute $L_0[\tau(j)] = \bar{g}_j$ for each $\bar{g}_j$, *i.e., exactly one node in each row* L_k *(k =0,1…,m-1). Alone, the segment tree therefore requires at least n*$\log_2$*n multiplications to compute* $\{\bar{g}_j\}$.

Results

Appendix B gives the combinatorics relevant to the Introduction's circRNA-miRNA application. As is typical in combinatorial probability, the quantities $\{g_i[k]\}$ *were combinatorial counts proportional to probabilities and spanned several orders of magnitude:* $\min_{i,k} g_i[k] = 1$ *and* $\max_{i,k} g_i[k] \approx 10^{452.8}$. *In Eq (1), the dimension K controls the number of terms in the convolutions. In the application, over each miRNA motif examined, the maximum number of motif occurrences on the circRNAs was K=997 . An Intel Core i7-3770 CPU computed the p-value relevant to the biological application on June 17, 2015. To compare later with estimated times for competing algorithms, the Jackknife Product algorithm with n=3086 computed the relevant p-values in about 45 minutes, requiring about 3n products. In the application, therefore, n products required about 15 minutes.*

The application of this article to circRNA-miRNA data appears elsewhere [1].

Discussion

This article has presented a Jackknife Product algorithm, which applies to any commutative semi-group $(G, \circ)$ *. The biological application to a circRNA-miRNA system exemplifies a general statistical method in combinatorial probability. In turn, the application in combinatorial probability exemplifies an even more general statistical test for whether a term in a sum of independent counting variates (not necessarily identically distributed) is unusually large. The Benjamini-Hochberg procedure for controlling the false discovery rate in multiple tests applies to independent variates [8] or to dependent variates with a positive regression dependency property [9]. Inconveniently, however, independent summands are negatively correlated when conditioned on a fixed sum. In the circRNA-miRNA application, therefore, the Benjamini-Hochberg procedure was unavailable to correct for multiple testing. Fortunately, a Bonferroni multiple test correction [10] sufficed because empirically there, any p-value not almost 1 was extremely small.*

*The Results state that for n = 3086, the Jackknifed Product algorithm computed the relevant p-values in about 45 minutes, with n products requiring about 15 minutes of computation. In contrast, the naïve algorithm avoiding inverses and requiring n (n-2) products would have taken about 3086*15 minutes, i.e., about 1 month. As explained under the "Computational Time and Storage" heading in the Theory section, without exploiting the special form of the jackknife products* $\{\bar{g}_j\}$ *, a segment tree requires about n products for its construction and at least* $n \log_2 n$ *products for the computation of the products* $\{\bar{g}_j\}$ *. Alone, segment tree algorithms would therefore have taken a minimum of about (1+log<>sub23086)*15 minutes, i.e., about 3 hours.*

Eq (1) might suggest that jackknife products are susceptible to computation with Fourier or Laplace transforms. The Results section notes that in the biological application, however, $\{g_i[k]\}$ *in Eq. (1) spanned several orders of magnitude, from* $\min_{i,k} g_i[k] = 1$ *to* $\max_{i,k} g_i[k] \approx 10^{452.8}$ *. On one hand, the widely varying magnitudes necessitated an internal logarithmic representation of* $\{g_i[k]\}$ *in the computer, a minor inconvenience for direct computation with the Jackknife Product algorithm. On the other hand, they might have presented a substantial obstacle for transforms. The famous Feynman anecdote about Paul Olum's (*10^{100}*) problem indicates the reason [11]:*

So Paul is walking past the lunch place and these guys are all excited. "Hey, Paul!" they call out. "Feynman's terrific! We give him a problem that can be stated in ten seconds, and in a minute he gets the answer to 10 percent. Why don't you give him one?" Without hardly stopping, he says, "The tangent of 10 to the 100th." I was sunk: you have to divide by pi to 100 decimal places! It was hopeless.

The Jackknife Product algorithm also abstracts to any commutative semigroup $(G, \circ)$ *, broadening its applicability enormously. As usual, abstraction eases debugging. Consider, e.g., the commutative*

semigroup consisting of all bit strings of length n under the bitwise "or" operation. If the bit string g_j has 1 in the j-th position and 0 s elsewhere, then the segment product $g_{[i,j)}$ equals the bit string with 1 s in positions $[i,j)=\{i,i+1,\ldots,j-1\}$ *and 0 s elsewhere. Similarly, the complementary segment product* $g_{\overline{[i,j)}}=g_{[0,i)}g_{[j,n)}$ *equals the bit string with 0 s in positions* $[i,j)=\{i,i+1,\ldots,j-1\}$ *and 1 s elsewhere. The Jackknife Product algorithm is easily debugged with output consisting of the segment and complementary segment trees for the bit strings.*

As a final note, even if a semigroup $(G,\circ)$ *lacks the Commutative property, the general product algorithm for a segment tree can still compute* $\{\overline{g}_j=g_{[0,j)}g_{[j+1,n)}\}$*. In a commutative semigroup* $(G,\circ)$ *,however, the downward phase of the Jackknife Product algorithm exploits the special form of the products* $\{\overline{g}_j\}$ *to decrease the time to O(n).*

Conclusions

This article has presented a Jackknife Product algorithm, which applies to any commutative semi-group $(G,\circ)$ *.The biological application to a circRNA-miRNA system uses a commutative semigroup of truncated convolutions to exemplify a specific application to combinatorial probabilities. In turn, the specific application in combinatorial probability exemplifies an even more general statistical test for whether a term in a sum of independent counting variates (not necessarily identically distributed) is unusually large. The abstraction from convolutions to commutative semi-groups broadens the algorithm's applicability even further. If an application only requires jackknife products* $\{\overline{g}_j\}$ *and their number n is large enough, the Results and Theory sections show that the linear time of the Jackknife Product algorithm can make it well worth the programming effort.*

Declarations

Availability of Data and Materials

A self-testing, annotated file "jls_jackknifeproduct.py" implementing an in-place Jackknife Product algorithm in Python is available without any restrictions at https://github.com/johnlspouge/jackknife-product . Data were previously published elsewhere [1].

Competing Interests
The authors declare that they have no competing interests.
Funding

This research was supported by the Intramural Research Program of the NIH, National Library of Medicine, and by the Center for Cancer Research, National Cancer Institute.

Authors' Contributions

JLS developed the algorithms, performed computational and statistical analysis, and drafted the manuscript. JMZ provided the data stimulating and exemplifying the analysis. MG performed bioinformatics and sequence analysis. All authors read and approved the final manuscript.

Acknowledgements

JLS would like to acknowledge useful conversations with Dr. Amir Manzour and Dr. DoHwan Park.

References

1.
Tagawa T, Gao SJ, Koparde VN, Gonzalez M, Spouge JL, Serquina AP, Lurain K, Ramaswami R, Uldrick TS, Yarchoan R, et al. Discovery of Kaposi's sarcoma herpesvirus-encoded circular RNAs and a human antiviral circular RNA. Proc Natl Acad Sci U S A. 2018;115(50):12805–10.

2.
Siegel S. Nonparametric Statistics for the Behavioral Sciences. 1 ed. New York: MacGraw-Hill; 1956.

3.
Freeman GH, Halton JH. Note on an exact treatment of contingency, goodness of fit and other problems of significance. Biometrika. 1951;38(1–2):141–9.

4.
Fisher Exact Test Batch Processing [https:// tinyurl.com/spouge-fisher] .

5.
Efron B, Stein C. The Jackknife Estimate of Variance. Ann Stat. 1981;9(3):586–96.

6.
Artin M: Algebra. Eaglewood Cliffs NJ: Prentice-Hall; 1991.

7.
Laaksonen A. Competitive Programmer's Handbook; 2017.

8.
Benjamini Y, Hochberg Y. Controlling the False Discovery Rate - a Practical and Powerful Approach to Multiple Testing. J R Stat Soc Ser B-Methodol. 1995;57(1):289–300.

9.
Benjamini Y, Yekutieli D. The control of the false discovery rate in multiple testing under dependency. Ann Stat. 2001;29(4):1165–88.

10.
Hochberg Y. A sharper Bonferroni procedure for multiple tests of significance. Biometrika. 1988;75(4):800–2.

11.
Feynman R, Leighton R, Hutchings E. Surely your're joking, Mr. Feynman! New York: W.W. Norton & Company; 1985.

Figures

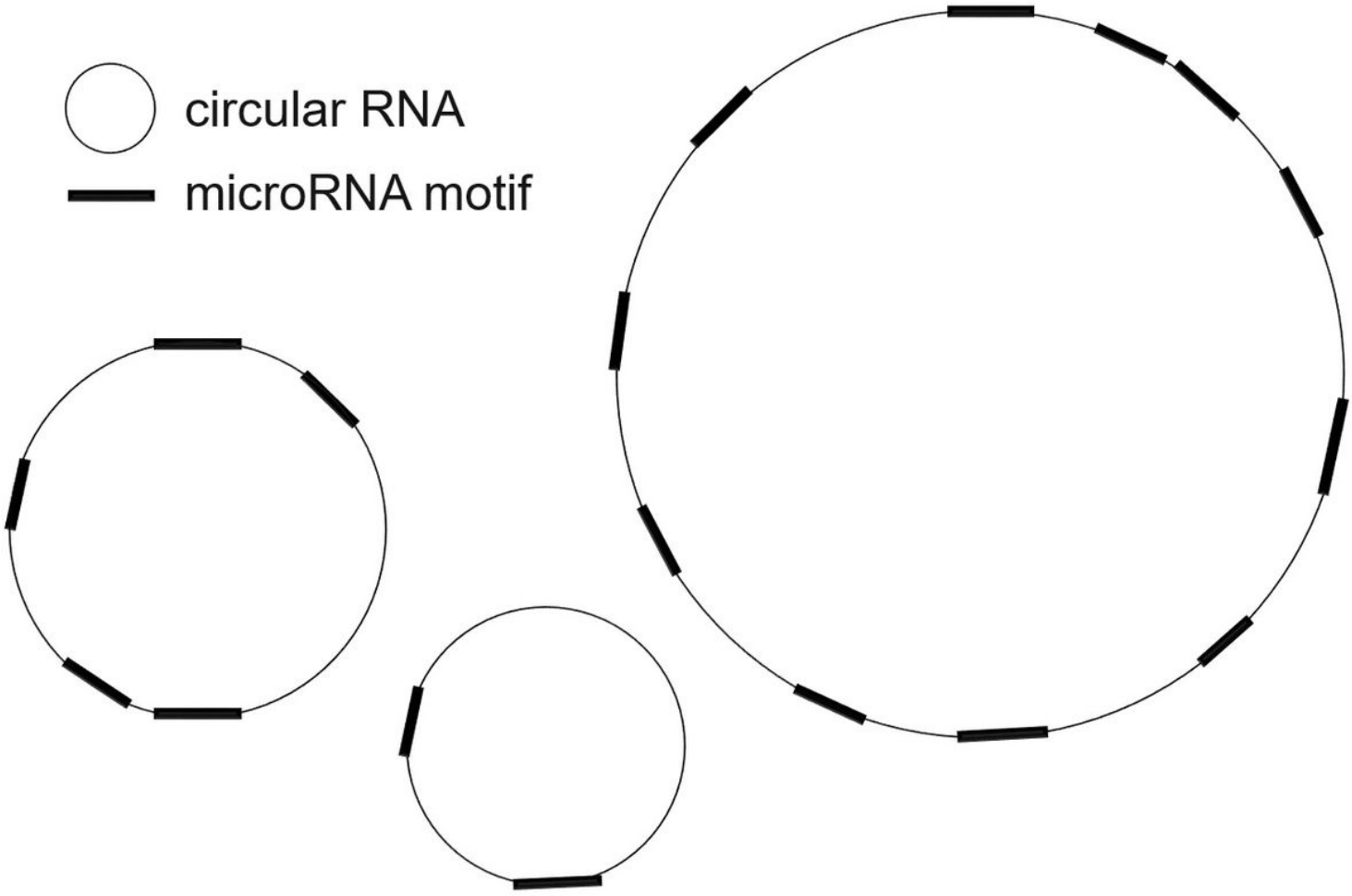

Figure 1

A schematic diagram of herpesvirus miRNA motif occurring on a human circRNA As indicated in the legend, each thin circle represents a circRNA; each thick line segment, the occurrence of a miRNA motif on the corresponding circRNA. Both circRNAs and the miRNA motif have nucleotide sequences represented by IUPAC codes {A, C, G, U}. Figure 1 illustrates occurrences of a single miRNA motif (e.g., UUACAGG) on the circRNAs. The biological question is: "does any circRNA have too many occurrences of the motif to be explained by chance alone?" In the actual application, the n = 3086 circRNAs ranged in length from 69 nt to 158565 nt.

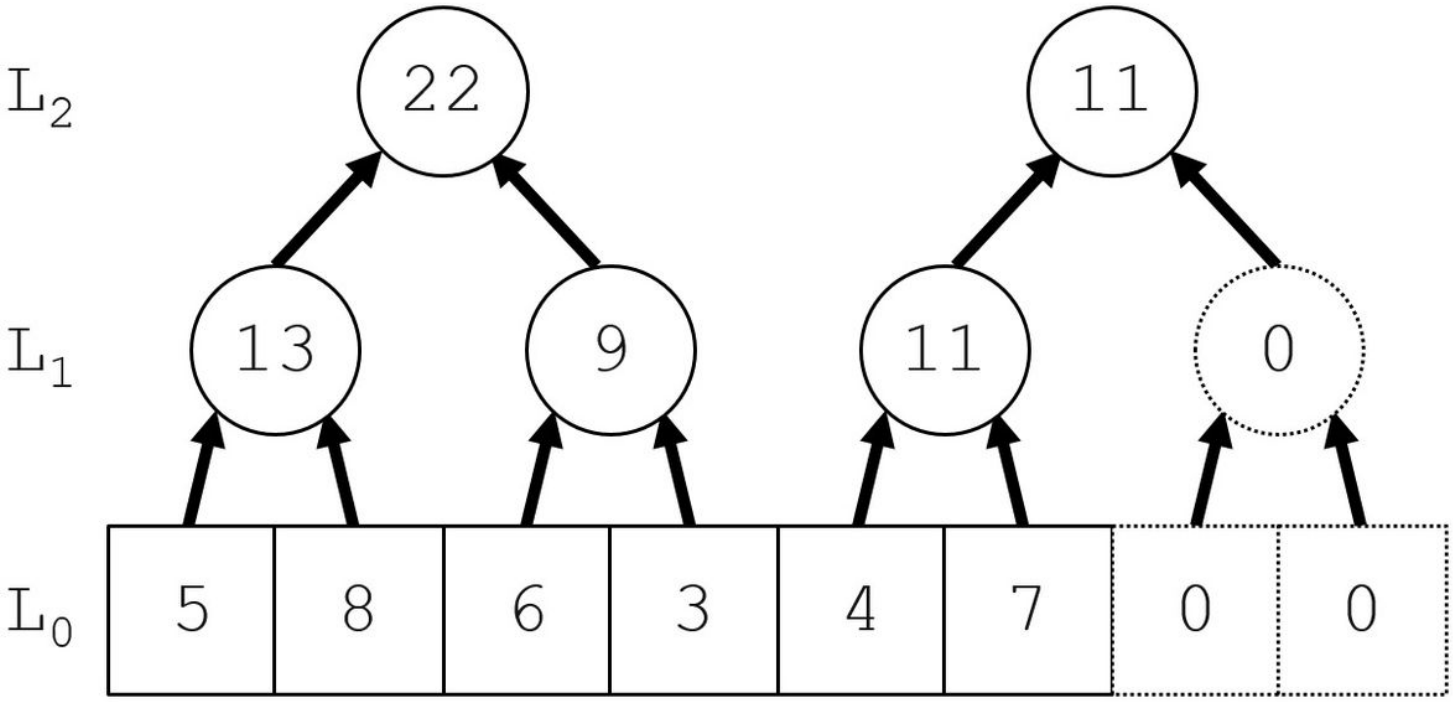

Figure 2

A (rootless) segment tree Figure 2 illustrates the rootless segment tree constructed in the upward phase of the Jackknife Product algorithm. The commutative semigroup (G, o) illustrated is the set of nonnegative integers under addition. The bottom row of n = 2m squares (m=3) contains L0[j]=gj (0<J<n*). In the next row up, as indicated by the arrow pairs leading into each circle, the array L1 contains consecutive sums of consecutive disjoint pairs in L0 , e.g., L1[0] =13=5+8. The rest of the segment tree is constructed recursively upward to Lm-1 , just as L1 was constructed from L0 . Here, 2 copies of the additive identity e = 0 pad out L0 on the right. Padded on the right, the copies contribute literally nothing to the segment tree above them. Their non-contributions have dotted outlines.*

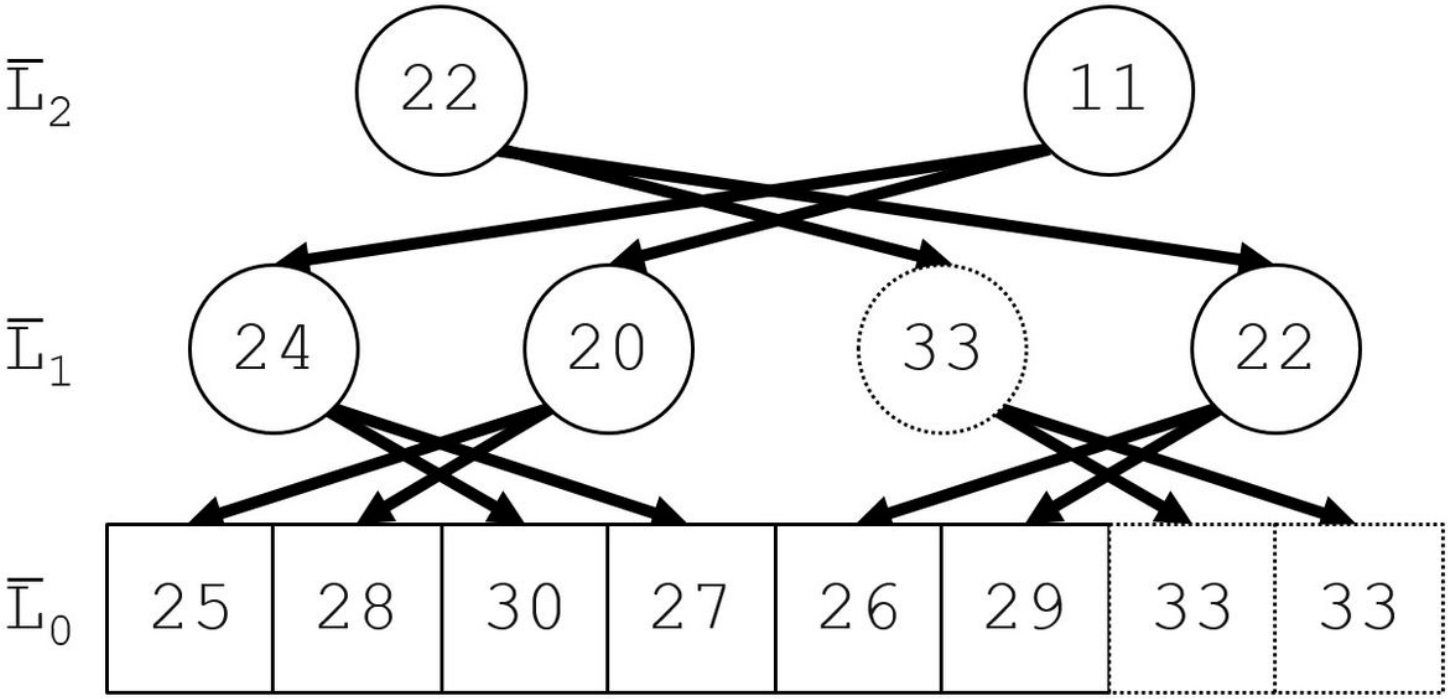

Figure 3

A (rootless) complementary segment tree Figure 3 illustrates the rootless complementary segment tree constructed in the downward phase of the Jackknife Product algorithm from the rootless segment tree in Figure 2. The downward phase starts by initializing the topmost row Lm-1 (m=3) with the topmost row Lm-1 of the rootless segment tree. The row L2 in Figure 2 and the row L2 in Figure 3, e.g., contain 22 and 11. For each Lk-1[j] in Figure 3, downward arrows run from Lk[αk (j)] to Lk-1[j] . As they indicate, each node in Lk contributes to its 2 "nieces" in Figure 2 to produce the next row down in Figure 3, e.g., L2[1] =11 contributes to its nieces l1[0]=13 and L1[1] =9 in the segment, to produce L1[0]=13+11=24 and L1[0]=9+11=20 in the complementary segment tree. The rest of the complementary segment tree is constructed recursively downward to L0 , just as L1 was constructed from L2 . In Figure 2, the elements of (in squares) total 33. To demonstrate the effect of the Jackknife Product algorithm, subtract in turn in Figure 3 each element (25, 28, 30, 27, 26, 29, 33, 33) in the bottom row from the total 33. The result (8, 5, 3, 6, 7, 4, 0, 0) is the bottom row L0 in Figure 2 with successive pairs transposed, so L0[j]= gτ(j), or equivalently gj=L0[τ(j)] .

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- *Appendix.docx*