Journal of
Bioinformatics and Comparative Genomics

Research Open Access

Optimum-time, Optimum-space, Algorithms for k-mer Analysis of Whole Genome Sequences

Sumedha S. Gunewardena

Department of Molecular and Integrative Physiology, Department of Biostatistics,
University of Kansas Medical Center, Kansas City, Kansas, USA

***Corresponding author:** Sumedha S. Gunewardena, Department of Molecular and Integrative Physiology, Department of Biostatistics, University of Kansas Medical Center, Kansas City, Kansas, USA; E-mail: sgunewardena@kumc.edu

Received Date: November 21, 2013 **Accepted Date:** January 25, 2014 **Published Date:** January 30, 2014

Citation: Sumedha S. Gunewardena (2014) Optimum-time, Optimum-space, Algorithms for k-mer Analysis of Whole Genome Sequences. J Bioinfo Comp Genom 1: 1-12.

Abstract

The sizable amount of data generated by high throughput cell biology is increasing the demand on traditional computational tools in bioinformatics to handle large input datasets. Large sequence data sets create intractable search spaces that are beyond the scope of many conventional algorithms. One way to address this problem is to transform large sequence data sets to the constituent parts that characterize the features of interest (e.g. transcription factor binding sites, miRNA sites, etc.) of the problem. These features of interest take the form of k-mers in a large subset of problems in computational biology. K-mers also play an implicit role in many other bioinformatics functions from microarray probes to genomic compositional analysis.Given the increasing potential for k-mers in a wide spectrum of applications in bioinformatics we present in this paper a set of fast and efficient generic algorithms for enumerating the occurrence frequen-cies of all substrings of a given length (k-mers) in whole genome sequences. Described are three algorithms of increasing complexity designed to deal with different k-mer lengths from short (couple of bases) to very long (tens of thousands of bases). They are memory based algorithms that use advanced heuristics to efficiently process large amounts of data that arise when analyzing very long genome sequences. The algorithms were tested for performance on the human, mouse, 681 bacteria and 50 archaea genome sequences. Results are described for both time and space utilization. We also describe several different experiments that demonstrate the utility of these algorithms. These algorithms can be downloaded from http://www2.kumc.edu/siddrc/bioinformatics/publication.html.

Introduction

K-mers play an implicit but very important role in many applications in computational biology as they form the characterizing unit of many interesting DNA sequences. For example, a transcription factor can be represented as a subset of 8-mers or 10-mers or some other k-mer, microarray probes can be defined as a collection of 25-mers, or some other suitable k-mer of choice, etc. It makes analysis of these sequence features very fast and very efficient if we can breakdown very long sequences (from for example Chip-sequence data or whole genome sequences) into their representative k-mers of interest. While on one hand this provides a lot of information on the relative abundance of different k-mers (useful for example in tasks such as generating microarray probes), it also enables us to drastically reduce the search space for applications such as motif detection (especially important when scanning for motifs in hundreds of sequences with lengths over thousands of bases). The problem is not trivial when relatively large k-mers in long DNA sequences are sought. The algorithms that we present in this paper are generic and can easily assist with or incorporate to any one of the many applications that utilize k-mers in their analysis.With the advent of large-scale genome sequencing projects (over 4600 completed or ongoing genome sequencing projects worldwide, [1]), there has being an exponential growth in the number of whole genome sequence data added to the literature (over 900 fully sequenced genomes to date, [1]). With this growth comes an increasing demand for efficient computational tools for analyzing this data. Counting the number of unique k-mers in a sequence is one such tool widely used in genomic analysis. k-mer analysis has wide ranging applications varying from whole genome compositional analysis to comparative genomics. Many nucleotide sequences of interest such as transcription factor binding sites, ribosome binding sites, microRNA sites, restriction sites, splice sites, primer sequences, probe sequences, etc. can be characterized by k-mers.

A k-mer is a string of length k over an alphabet Σ, where Σ is the four nucleotide bases {A; C; G; T}, for DNA sequences. In traditional k-mer analysis, k varies between 1 and 12. However, we are interested in unrestricted values of k making the problem much more computationally challenging in the management of both time and space (memory). k-mer analysis involves computing the frequencies of all distinct sub-strings of length k appearing in an analyzed sequence. While k-mer analysis using system memory forsmall values of k on short sequences of several thousand bases can be accomplished with relative ease (see [2], where the authors describe a tool AS-SIRC which uses hashing functions to find fixed sized regions of similarity which are then extended by a random walk procedure), the problem becomes challenging when long k-mers on whole genome sequences with lengths over a million bases in a single chromosome accumulating to over a billion bases in total are considered. Existing algorithms deal with longer k-mers by using external memory (see [3], which describes the tool REPuter which computes maximal repeats of an input sequence. It does this by building a suffix tree of the input sequence and performing a depth first traversal of the tree to locate all the nodes representing maximal repeats) which is unlike the case of the proposed algorithms which work only with internal memory.

In this paper we describe three algorithms with increasing complexity for counting k-mers. The first of these algorithms is designed for high-level programming languages (such as Matlab and R) where system memory is considerably restricted in both capacity and organization. This algorithm, depending on the available system memory, works efficiently for small values of k (up to 8 to 12) with no restriction on the length of the input sequence. The second algorithm builds on the conceptual framework of the first but allocates memory parsimoniously. As a result, it is able to compute counts for larger values of k (up to 16). The third algorithm uses heuristics to extend the functionality of the second algorithm to potentially unrestricted values of k. The striking feature of all three proposed algorithms is that they work with internal system memory as opposed to external memory as is the general case with algorithms that have to deal with very large data sets (see [4, 5], which describes SequeX, a tool that uses static SB-trees that are indexed data structures for external memory for analyzing k-mers). As a result they are very fast and convenient to amalgamate with different applications. Other algorithms are limited in the k-mer length they could analyze (see [6] which describes the program Jellyfish which is based on a multithreaded lock-free hash table. It uses a bit-packed data structure for memory efficiency. This algorithm is however limited to a maximum k-mer length of 31 bases) which is not the case with the proposed algorithms.

Results

Algorithm 01: A Linear-Time Algorithm for short k-mer Analysis of Whole Genome Sequences

The first algorithm that we describe computes frequencies of all b^k different strings (where b is the cardinality of the alphabet Σ) of length k in a sequence of length n in $O(n)$ time. However, as it employs a constant linear array of b^k elements to store the b^k different frequencies, it is encumbered by available memory as k grows. As a result, this algorithm is only suitable for computing k-mer frequencies for small values of k (up to k=12), and is principally proposed as an efficient implementation in high level programming languages (such as Matlab and R) that lack the conveniences of flexible dynamic memory allocation routines.

The proposed algorithm is a variation of the Rabin and Karp [7] stringing matching algorithm. It maps each nucleotide to an integer in radix-b and uses this integer representation to uniquely index each k-mer pattern as a base b number where b = |Σ|. For example, if the nucleotides {A; C; G; T} are transformed using the transformation (A7→0; C7→1; G7→2; T7→3) to give a corresponding integer representation in radix-4 notation, we can view every k-mer (and its reverse compliment) as a length k number in base-4 (see Figure 1. (a)). The idea is to compute the index of each successive transformed k-mer (and its reverse compliment) in O(1) time. This idea is made precise in the k-merCount algorithm described in Methods.

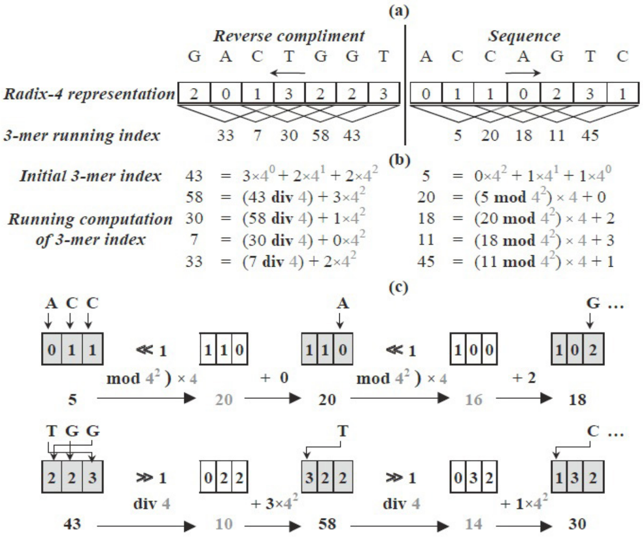


Figure 1. Computation of indices by the k-merCount algorithm. The figure illustrates the computation of successive indices of every 3-mer and its reverse complement of the sequence ACCAGTC. (a) The radix-3 representation of the sequence and its reverse compliment and the indices of each successive 3-mer and its reverse compliment of the sequence. (b) Computation of successive indices. (c) Graphical illustration of the computation of the index of a 3-mer and its reverse compliment in constant time given the index and reverse compliment index of its predecessor.

Algorithm 02: An Optimal-Time Algorithm for k-mer Analysis

The second algorithm described below called GK-MER-COUNT can compute k-mer frequencies for larger values of k (up to k=16 in a computer with 2Gb of internal memory but could deal with larger values of k with the increase in internal memory or for smaller genomes) than the k-mer-Count algorithm. It achieves this by parsimoniously allocating space when storing individual k-mer counts. Instead of storing frequencies of all b^k different k-mers as the k-mer-Count algorithm does, this algorithm only stores those k-mer frequencies (and their reverse compliments) that appear in the input sequence. The GK-MER-COUNT algorithm takes advantage of the fact that although there are b^k possible patterns of length k, there could only be at most $n-k+1$ different patterns in a sequence of length n, hence, the actual number of different patterns, $M \leq \min(b^k; n-k+1)$, is linearly bound for large values of k.

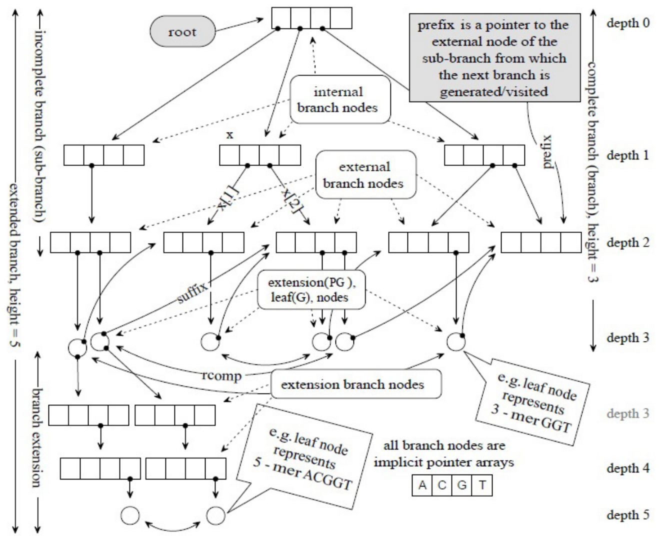

root
prefix is a pointer to the external node of the sub-branch from which the next branch is generated/visited
internal branch nodes
external branch nodes
x
x[1]
x[2]
prefix
suffix
extension(PG), leaf(G), nodes
rcomp
extension branch nodes
e.g. leaf node represents 3 - mer GGT
e.g. leaf node represents 5 - mer ACGGT
all branch nodes are implicit pointer arrays
A C G T
depth 0
depth 1
depth 2
depth 3
depth 3
depth 4
depth 5
incomplete branch (sub-branch)
extended branch, height = 5
branch extension
complete branch (branch), height = 3

Figure 2. The k-mer tree. The figure illustrates the components and terminology associated with a k-mer tree.

The GK-MER-COUNT algorithm uses a special dynamic tree data structure which we call a *k-mer tree* (see Figure 2.) for holding the frequencies of all sub-sequences of a specified length occurring in agiven sequence. A k-mer tree is a modified full n-array tree where n = |Σ| = b, (for example, n = 4 for Σ = {A, C, G, T}). A k-mer tree for counting the frequencies of all length *k* sub-strings of a sequence will have height k. Every branch node of a k-mer tree is a *b* element array of pointers referring to either a branch node or a leaf node. Each element of the array (branch node) implicitly refers to a unique element in Σ by way of its *index* (e.g. *index*(0) ≡A, *index*(1) ≡C, *index*(2)≡ G, *index*(3)≡T). In a k-mer tree, a complete branch or simply a branch is a branch that starts from the internal branch node at depth 0 and ends in a *leaf node* at depth *k*. An incomplete-branch or a sub-branch of the tree is a branch that starts from the internal branch node at depth 0 and ends in an external branch node at depth *k*– 1 and hence not complete (does not have a *leaf node*). A k-mer tree has only complete and incomplete branches as defined above. Every branch of a k-mer tree represents a unique k-mer whose sequence is read implicitly from the index of the pointers along its path. Every *leaf node* of the tree has three essential elements, a counter to hold the frequency of its representative k-mer, a suffix pointer which is set to point to the external branch node of the sub-branch forming its length *k*– 1 suffix and a reverse compliment pointer, rcomp, set to point to the leaf-node of its reverse compliment.

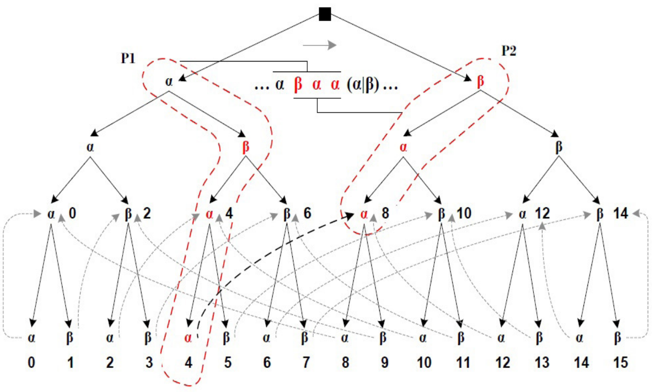
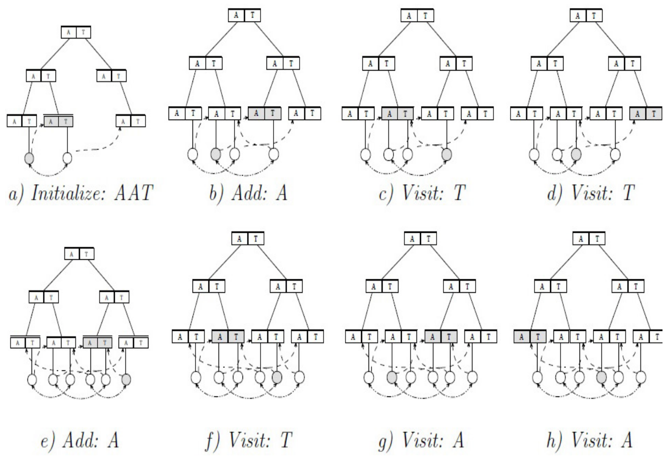


Figure 3. A full 4-mer-tree. A full 4-mer-tree for counting the frequencies of distinct 4-mers in a sequence comprising of characters, and . As the tree is a complete 4-mer tree, it has only complete branches. Every branch represents a distinct 4-mer. Every *leaf node* points to an external branch node that is the external branch node of the sub branch representing the suffix of the 4-mer along the branch to the *leaf node*. In analogy to the k-mer-Count algorithm described in Section , moving from a *leaf node* to the external branch node represents a shift to the left by one position of the radix b digit of length k (in this case, a binary digit of length 4). Proceeding down to a *leaf node* is analogous to adding a new digit as a lowest order digit of the k-digit radix-b number (4-digit binary number).

The GK-MER-COUNT algorithm is conceptually similar to the k-mer-Count algorithm described above. This is demonstrated in Figure 3. on a binary alphabet, Σ = {;}, for counting the frequencies of all 24, 4-mers of a sequence. The GK-MER-COUNT algorithm uses a circular buffer (of length k) to keep track of successive k-mers as characters are read from the input sequence thereby maintaining a linear read time while still retaining access to the full k-mer sequence at a particular instance. The complete GK-MER-COUNT algorithm is described in detail in Methods. However, in brief, the algorithm works as follows: Branches of a k-mer tree represent distinct k-mers. Adding or completing a branch in a k-mer tree in the context of the GK-MER-COUNT algorithm involves 1) adding the complete branch representing the k-mer (from the root node to the *leaf node*) or completing the branch by adding a leaf to the appropriate external branch node, 2) adding the sub branch representing the *k*– 1 suffix of the k-mer, 3) adding the branch representing the reverse compliment of the k-mer, and, 4) adding the sub branch representing the *k*– 1 suffix of the reverse compliment. Sequence information of the k-mer necessary for this task is red from the circular buffer mentioned above. A pointer is established from the *leaf node* of the k-mer branch to the external branch node of its suffix sub branch. Similarly, a pointer is established from the *leaf node* of the reverse compliment branch to the external branch node of the reverse compliment's suffix sub branch. A pointer is also established between the *leaf node*s of the k-mer branch and its reverse compliment branch. The GK-MER-COUNT algorithm maintains a *prefix* pointer that always points to the external branch node of the sub branch that forms the *k*– 1 *prefix* of the next k-mer.

In the initialization step, the first branch of the k-mer tree is added representing the first k-mer of the input sequence. The *prefix* pointer is directed to the external branch node of its suffix sub branch. Note that the *prefix* pointer now points to the external branch node of the *prefix* sub branch of the next k-mer. Now, at a particular instance of the algorithm, when the next character is read, the algorithm checks whether the *leaf node* for the read character exists in the external branch node pointed to by the *prefix* pointer. If the *leaf node* does exist (the k-mer is already in the tree), the algorithm simply increments its frequency counter and that of its reverse compliment (using the rcomp pointer) and using the information in the suffix pointer of the *leaf node*, moves the *prefix* pointer to the external branch node of the suffix sub branch of the k-mer. If the *leaf node* does not exist (the k-mer has not yet being added to the tree), it is added along with the sub branch representing the $k - 1$ suffix of the k-mer, the branch representing the reverse compliment of the k-mer and the sub branch representing the $k - 1$ suffix of the reverse compliment. As before, pointers are established from the *leaf node*s to the respective external branch nodes of the suffix sub branches. A pointer is also established between the *leaf node*s of the complimentary branches. The *prefix* pointer is moved to the external branch node of the suffix sub branch of the k-mer. A simple example underlying the above process is illustrated in Figure 4.



a) Initialize: AAT *b) Add: A* *c) Visit: T* *d) Visit: T*

e) Add: A *f) Visit: T* *g) Visit: A* *h) Visit: A*

Figure 4. Example: step by step creation of a k-mer tree. The figure demonstrates the creation of a k-mer tree for counting the frequencies of different 3-mers in sequence AATATTATAA. For simplicity we assume Σ = {A; T }. In the full 2-ary tree that is generated, we assume that, the left cell of a branch node represents A and the right cell represents T (in practice, these will be pointers with only implicit reference to the nucleotide). Every time a *leaf node* is added or visited, the counter for its respective k-mer and that of its reverse compliment are incremented by one. In the initialization step in (a), the branch AAT is added followed by the sub-branch of its suffix AT. A pointer is established from the *leaf node* of AAT to the bottom node (external branch node) of the sub-branch AT. The branch ATT, which is the reverse compliment AAT, along with the sub-branch of its suffix TT are added next. A pointer is established between the *leaf nodes* of the complimentary branches AAT and ATT. The *prefix* pointer (the pointer pointing to the external branch node to which the next *leaf node* will be added, or, from which the next *leaf node* will be visited) is moved to the external branch node of sub-branch AT. In (b), the *leaf node* A is added from the external branch node of sub-branch AT establishing the branch ATA. Similar to (a), its suffix TA and reverse compliment TAT are added. The *prefix* pointer is moved to the sub-branch TA. In (c) *leaf node* T is visited as it is already in sub-branch TA, (incrementing the frequency counter of 3-mer TAT and its reverse compliment ATA), and the *prefix* pointer is moved to the sub-branch AT. In (d), similar to (c), the *leaf node* T is visited and the *prefix* pointer is moved to the sub-branch TT. In (e), *leaf node* A is added establishing the 3-mer, TTA, and its reverse compliment TAA. The *prefix* pointer is moved to the sub-branch TA. Like above, in (f), *leaf node* T is visited. In (g), *leaf node* A is visited and in (h) *leaf node* A is visited.

The GK-MER-COUNT algorithm has a running time of Θ(M (k – 1) + n), where M is the actual number of distinct k-mers in the input sequence. At its worst case, the algorithm has O(nk) running time. This occurs when every length k sub sequence of the input sequence is distinct, i.e. M = n. However, in practice M is a lot smaller than n due to multiple occurrences of distinct k-mers. On average, given an equal nucleotide distribution on the input sequence, one would expect to see n=4k hits per k-mer. According to this statistic the expected number of multiple occurrences per distinct k-mer decrease exponentially with the increase in k resulting in larger trees. This problem is addressed in the next algorithm described below.

A complete k-mer tree, i.e., a k-mer tree representing all b^k distinct k-mers will have b^k *leaf node*s and (b^k– 1)=3 branch nodes. This is space wise manageable for small values of k. As explained above, for large values of k, the actual number of distinct k-mers, M , in the input sequence is linearly bound therefore the size of the k-mer tree is upper bound by the length of the input sequence. In the worst case, the GK-MER-COUNT algorithm is linear in space (see Figure 5 (A)). The GK-MER-COUNT algorithm is suitable for counting the k-mer frequencies of DNA sequences for moderate values of k (up to k =16 in a machine with 2Gb of internal memory but could support larger k in machines with larger memory or for smaller genomes).

Algorithm 03: A Heuristic Algorithm for Large k-mer Analysis

The third algorithm, called PGK-MER-COUNT , is designed to count k-mer frequencies for large values of k. It is a heuristic algorithm that works on the understanding that the desired k-mers are those that appear with high frequency in the analyzed DNA sequence. The expected frequency of a given k-mer (n=4^k) decreases exponentially with the increase in k. As a result the frequency of a majority of k-mers of a sequence will be one or very small for large values of k. On the other hand, space required to store different k-mer counts increases exponentially with the increase in k, reaching a maximum when $k = n/2$, with O(n2) representations. This is extremely large even for small genome sequences of around a million bases with much if not all of the k-mers having a frequency of one. An investigator is usually interested in not these low frequency k-mers but those k-mers that appear with relatively high frequency in the analyzed sequence. The PGK-MER-COUNT algorithm can be used to obtain these k-mers very efficiently.

The PGK-MER-COUNT algorithm is described in detail in Methods but in brief the algorithm works as follows: The PGK-MER-COUNT algorithm performs two scans of the input sequence. The first scan performs the GK-MER-COUNT algorithm described above with a sub k-mer length, k_0, that is much smaller than the required k-mer length k. The second scan uses the frequency counts of the sub k-mers computed in the first scan to heuristically extend the k-mer tree to accommodate the frequency counts of the desired k-mers (of length k) that appear with a frequency greater than or equal to a given threshold value, in the scanned sequence. The value is the minimum desired k-mer frequency of interest.

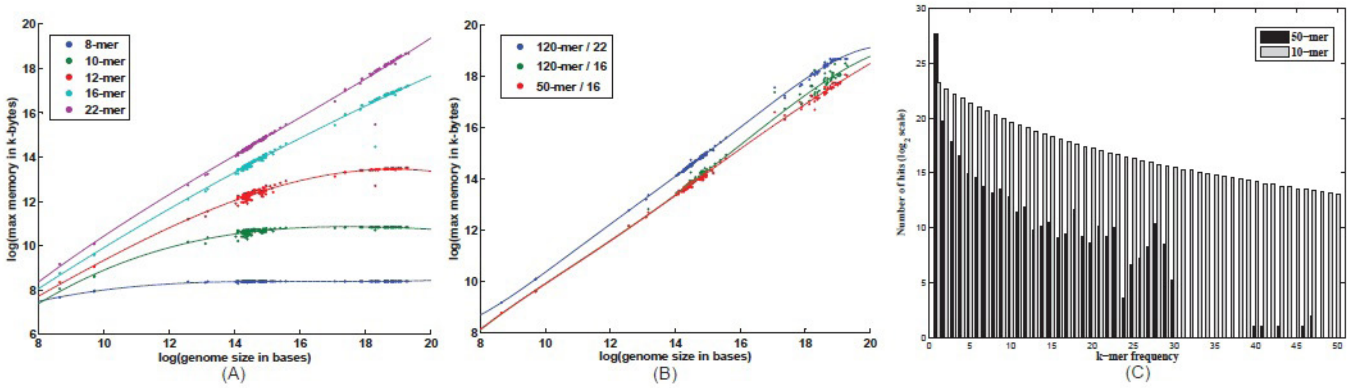


Figure 5. Memory utilization of the P/GK-MER-COUNT algorithms. Memory utilization of the (A) GK-MER-COUNT algorithm for different values of k and (B) PGK-MER-COUNT algorithm, with = 2, (k0; k) = (22; 120); (16; 120) and (16; 50), on the human, mouse and 50 archaea genome sequences listed in Supplemental Table 01. (C) Comparative histogram of 50-mer and 10-mer frequency counts.

In its second scan, the PGK-MER-COUNT algorithm extends the initial k-mer tree generated in the first scan (having k-mer length k_0) to accommodate the desired k-mers of length *k*. This is achieved by adding branch extensions to the complete branches of the initial k_0-mer tree (or simply incrementingthe k-mer count if the extended branch already exists). A branch extension is added if and only if the substring (of length *k*) representing the extended branch is encountered in the scanned sequence, *s*, with a minimum k_0 on k substring abundance greater than or equal to the threshold . The minimum k_0 on *k* substring abundance of a length *k* substring in sequence s is define as the minimum frequency in the k_0-mer tree of the $k–k_0+ 1$ substrings of length k_0 in the length k string. This is a heuristic extension based on the rational that all $k–k_0+ 1$ substrings of length $k_0< k$ of any substring of length *k* in sequence *s* with a frequency greater than or equal to the desired minimum frequency, , will havefrequencies greater than or equal to in *s*. Conversely, if any string of length k in sequence s contains a substring of length $k_0< k$ with a frequency less than *α*, then the string itself will have a frequency less than *α*. In other words the maximum frequency of a string of length k in sequence s is upper bounded by the minimum k_0 on k substring abundance of that string for some $k_0< k$. Based on this observation, we only need to add branch extensions to the k_0-mer tree if and only if the extension has a minimum k_0 on k substring abundance greater than or equal to the desired frequency.

The minimum k_0 on *k* substring abundance for all length *k* substrings in sequence *s* can be computed in linear time by traversing along the *leaf node*s of the k_0-mer tree using the *suffix* pointer. Hence the time complexity of the second scan is $O(n + m(k–k_0))$ where m is the number of branch extensions added to the k_0-mer tree and is bounded by $0 \leq m \leq (n–k + 1)/\alpha$. Generally, m will be small for large values of kand . The sub k-mer length k_0 of the initial k_0-mer tree plays an import part in the efficiency in building the subsequent k-mer tree. The larger the value k_0, the more efficient will be the building of the subsequent k-mer tree. However, the sub k-mer length k_0 is restricted by the constraints underlying the GK-MER-COUNT algorithm. Ideally, , which is the desired minimum k-mer frequency will be a value greater than the expected k0-mer frequency, $n/4k_0$. Depending on the available system resources, one can reduce subject to this constraint by increasing k_0.

The amount of memory required for the PGK-MER-COUNT algorithm consists of the memory required for the first scan performed by the GK-MER-COUNT algorithm which is linear and the additional memory required for the extensions added in the second scan which is also linear for large input sequences. The PGK-MER-COUNT algorithm also has a linear worst case space utilization (see Figure 5 (B)).

The Modified PGK-MER-COUNT Algorithm

If sufficient memory is available to accommodate an initial sub k-mer length such that $n/4k_0 < \alpha$ for $\alpha = 2$, we can use a simple extension to the PGK-MER-COUNT algorithm to output all k-mers (including those with frequency one) in very long input sequences for large values of k. This is made possible by observing that every sub-substring of length k in the input sequence with a minimum k_0 on *k* substring abundance less than $\alpha = 2$ has frequency one. Therefore, all length *k* sub-sequences encountered in the second scan of the PGK-MER-COUNT algorithm with a minimum k_0 on k substring abundance of one can be directly sent to output (with a frequency count of one) without any additional cost. No record of it needs to be kept in memory. In practice, sub k-merlengths (k_0) of 14 to 16 can handle input sequences of 100 million to a billion bases in length.

Performance and Evaluation

The performance of the algorithms were evaluated using several whole genome sequences. These include experiments on Human chromosomes (1, . . . , 22,X,Y), Mouse chromosomes (1, . . . , 19,X,Y), 681 bacteria genome sequences and 50 archaea genome sequences, downloaded from the Ensembl[8] and EMBL [9], nucleotide sequence databases. These evaluations were performed to ascertain the space and time utilization of the algorithms. The experiments were carried out on a Linux machine with a 1.0Ghz processor and 32Gb RAM.

Time and space utilization

In our first experiment we tested the GK-MER-COUNT and PGK-MER-COUNT algorithms on 50 archaea genome sequences to establish the level of performance of thesetwo algorithms on time and space utilization. The 50 archaea genome sequences, listed in Supplemental Table 01, ranged in length between 5.7 and 0.3 million bases with an average length of 2.1 million bases. In total they comprised of roughly 106:5 million

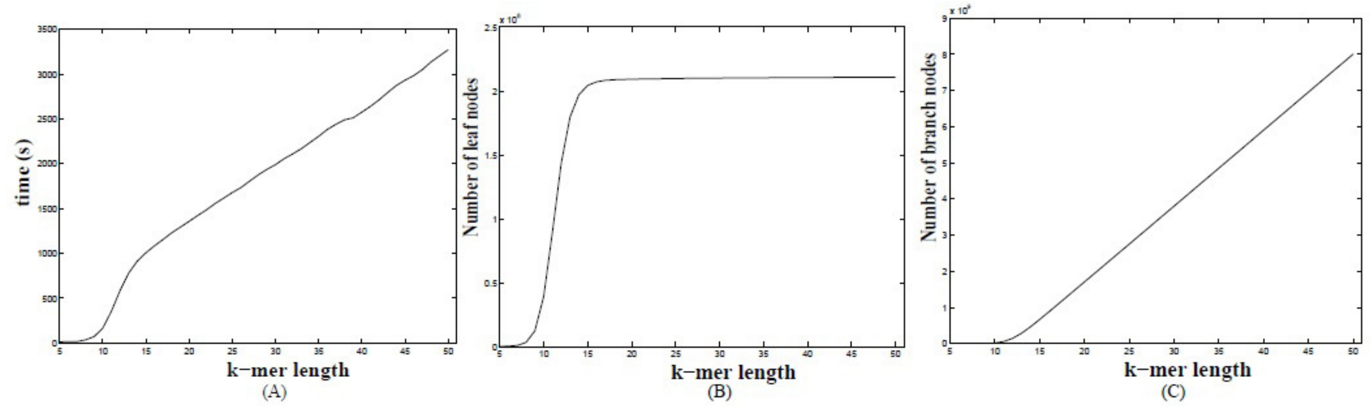


Figure 6. Performance of the GK-MER-COUNT algorithm. Performance of the GK-MER-COUNT algorithm on the 50 archaea genome sequences listed in Supplemental Table 01.

bases. As the algorithms simultaneously scan both the positive and negative strands of a given sequence, the actual sequence length scanned by them is double the input sequence length.

Figure 6. (A) shows the time taken by the GK-MER-COUNT algorithm to ascertain the k-merfre-quencies of the 50 archaea genome sequences as *k* varies from 5 to 50. It is evident that the algorithm performs fastest when k is small and increases linearly in time from around k = 15. There is a marked increases in time between k = 10 and k = 15. This rapid increase in time expended can be attributed to the sharp increase in the number of *leaf node*s generated by the algorithm between these two k-mer lengths as seen in Figure 6. (B). We can see from Figure 6. (B) that the number of *leaf node*s generated by the GK-MER-COUNT algorithm reaches an upper bound, which as explained above is determined by min(4^k; $n{-}k$ + 1). The linear increase in time, observed in Figure 6. (A), beyond k = 15, for computing the k-mer frequencies as k increases can be attributed to the linear increase in the number of branch nodes generated by the GK-MER-COUNT algorithm as k increases as seen in Figure 6. (C). As a testimony to the speed of the GK-MER-COUNT algorithm, we see from Figure 6. (A) that it can compute the 50-mer frequencies of 50 archaea whole genomes in less than an hour. It should be stated that a bulk of this time is spent on writing results to external storage.

Figure 7. (A) shows the running time of the PGK-MER-COUNT algorithm for computing the k-mer frequencies of the 50 archaea genome sequences described above, with a minimum desired k-mer frequency of 2. The algorithm was run with a sub k-mer length of 12. The running time of the PGK-MER-COUNT algorithm decreases exponentially with the increase in *k*. This is because the algorithm in generating Figure 7. (A) is only concerned with k-mers with frequencies greater than or equal to two eliminating the need to keep track of a vast number of k-mers having frequency one. This can be seen by looking at Figures 7. (B) and (C) which shows an exponential drop in the number of leaf and branch nodes generated by the PGK-MER-COUNT algorithm as *k* increase. Figure 5. (C) shows the difference in frequency counts between the 10-mer and 50-mer frequencies of the 50 archaea genome sequences described above. It can be seen that a substantial number of 50-mers have frequency one with the counts dropping rapidly with increasing frequency. The 10-mers on the other hand show a more uniform decrease in counts between adjacent frequencies with smaller differences between them. By avoiding low frequency counts, the PGK-MER-COUNT algorithm is able to make substantial savings in both time and space utilization. This savings increase exponentially with the increase in k-mer length.

Between genome composition analysis

An interesting biological exercise with a verity of ap-plications is the identification of genomes (from an array of bacteria genomes for example) having sub-sequences of a particular length that appear in low abundance in another genome (such as that of a newly sequenced species for example). Given a genome and a sub-sequence length of interest, being able to quickly and efficiently identify other genomes that have a low

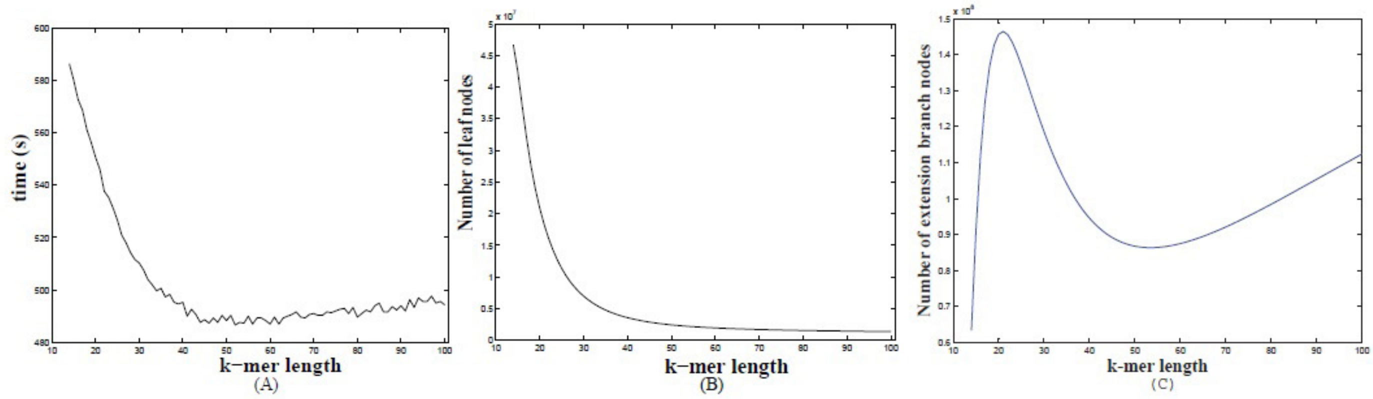


Figure 7. Performance of the PGK-MER-COUNT algorithm. Performance of the PGK-MER-COUNT algorithm, with = 2, on the 50 archaea genome sequences listed in Supplemental Table 01.

abundance of hits for sub-sequences from the first genome is a challenging but important problem in computational biology. To demonstrate the utility of the GK-MER-COUNT and PGK-MER-COUNT algorithms, we used them to gather statistics of the abundance of common 25-mers in the mouse chromosomes and 681 bacteria sequences. These results are shown in Supplemental Table 02. We used the PGK-MER-COUNT algorithm to obtain all 25-mers in the mouse chromosomes with a frequency of 2 or greater. We used the GK-MER-COUNT algorithm to obtain all 25-mers of the 681 bacteria sequences.

From this analysis we see that the mouse chromosome with the highest number of bacteria sequences that did not have any 25-mer that appeared at least twice was chromosome Y (528 bacteria sequences) and the lowest was chromosome 1 (340 bacteria sequences). There were 256 bacteria sequences without a single 25-mer that appeared at least twice in any of the mouse chromosomes. The largest of these was the soil bacterium 'SolibacterusitatusEllin 6076' with a chromosome length of 10.0 Mbp. There were 152 bacteria sequences that had at least one 25-mer that appeared at least twice in every mouse chromosome.

The largest of this was the bacterium '*Streptomyces avermitilis* MA-4680' with a chromosome of 9.0 Mbp and the smallest was the bacterium 'C*andidatus Sulcia muelleri* GWSS' with a chromosome length of 0.25 Mbp. There were 425 bacteria sequences that had at least one 25-mer that appeared at least twice in at least one of the mouse chromosomes. The bacterium 'Mycoplasma hyopneumoniae 7448' with a chromosome length of 0.92 Mbp had the highest number of unique 25-mers (3173) with at least two hits in a mouse chromosome.

Identifying longest duplicated segments

The PGK-MER-COUNT algorithm can find k-mer counts for very large values of *k* in very long input sequences. In order to validate this assertion, we used the PGK-MER-COUNT algorithm to find the longest duplicated sequence in each of the human chromosomes. In order to avoid interspersed repeats and low complexity DNA sequences, we used the masked version of the genome sequence of the human chromosomes obtained from Ensembl. The identified repeated segments for each chromosome are listed in Supplemental Table 03. The longest duplicated sub-string of an input sequence is given by the PGK-MER-COUNT algorithm when the input k-mer length *k* is such that it is the largest value for which a hit is noted when the algorithm is run with set to 2. This value of *k* can be easily found by iteratively running the PGK-MER-COUNT algorithm with different values of *k* until the value of k is found for which there is a hit but not for *k*+1.

From Supplemental Table 03.we see that human chromosome 11 has the longest duplicated sequence extending 5829 bases in spite of it being only the 12th largest in size. On the other hand, the longest duplicated sequence in chromosome 21 is 569 bases. It is interesting to note that after chromosome 11, it is the two sex chromosomes, chromosomes Y and X that have the longest duplicated sequence extending 5681 and 4720 bases respectively. Chromosome X is the most conserved human chromosome [10]. Chromosome 8 has two longest duplicated sequences of 3443 bases each. While the longest duplicated sequence of chromosomes 15 and 16 are in fact triplicates extending 3914 and 2777 bases respectively, the longest duplicated sequence of chromosome 13 is a quintuplicate extending 1261 bases.

Discussion

The GK-MER-COUNT and PGK-MER-COUNT algorithms are efficient algorithms for counting k-merfre-quencies of whole genome sequences. As they work with system memory in spite of the large amount of data they need to handle, they are very fast and convenient to use. They come at a time when an in-creasing number of newly sequenced genomic data are being added to the existing array of whole genome sequences creating a big demand for efficient tools to perform task of this nature.

Although the GK-MER-COUNT algorithm works with moderate length k-mers, it falls within the range of many of the k-mers of biological interest such as transcription factor binding sites, restriction sites, etc. The PGK-MER-COUNT algorithm on the other hand can gather k-mer statistics for large values of k from tens to thousands of bases in length serving the need of many applications that require k-merfrequency counts. Although the PGK-MER-COUNT algorithm is designed to gather high frequency k-mer counts for large k in long genomic sequences, with sufficient memory it can be used in its modified form to gather k-mer counts of all k-mers.

The k-mer-Count algorithm is the fastest amongst the three algorithms described above for small values of *k*. The GK-MER-COUNT algorithm is faster than the PGK-MER-COUNT algorithm for computing all frequency counts for moderate values of *k*. The modified PGK-MER-COUNT algorithm can sometimes outperform the GK-MER-COUNT algorithm for computing all frequency counts for moderate values of *k*, however its output is not ordered unlike that of the GK-MER-COUNT algorithm. The modified PGK-MER-COUNT algorithm on the other hand can compute all the frequency counts for large values of *k*. A majority of time expended by these algorithms is spent on writing output to external storage.

Methods

The k-mer-Count Algorithm

The k-mer-Count algorithm takes as input, a DNA sequence file *f*, the k-mer length *k*, and returns an array, *M* , of length $4^{k,}$ with the frequencies of all 4^k different patterns of length *k* in the DNA sequence and its reverse compliment. The algorithm uses the function getchar to read a character from *f*. The function convert, used in the algorithm, takes a nucleotide and returns its corresponding radix-4 digit. The function compliment returns the compliment of a converted nucleotide. Also used in the algorithm are the two functions div and mod that return the quotient and remainder of a division operation respectively. Arrays in the algorithm are one-based arrays.

```
K-MER-COUNT(f, k)

1  $h \leftarrow 4^{k-1}$
2  $I_w \leftarrow 0$
3  $I_c \leftarrow 0$
4  for $i \leftarrow 1$ to $k$
5        $c \leftarrow getchar(f)$
6        $I_w \leftarrow Iw + convert(c) \times 4^{k-i}$
7        $I_c \leftarrow Ic + compliment(\ convert(c)\ ) \times 4^{i-1}$
8  $M\ [I_w + 1] \leftarrow M\ [I_w + 1] + 1$
9  $M\ [I_c + 1] \leftarrow M\ [I_c + 1] + 1$
10 while ( $c \leftarrow getchar(f)$ ) $\neq eof$
11       $I_w \leftarrow (\ (I_w \bmod h\ ) \times 4\ ) + convert(c)$
12       $I_c \leftarrow (I_c\ div\ 4\ ) + compliment(\ convert(c)\ ) \times h$
13       $M\ [I_w + 1] \leftarrow M\ [I_w + 1] + 1$
14       $M\ [I_c + 1] \leftarrow M\ [I_c + 1] + 1$
15 return M
```

The k-mer-Count algorithm works as follows. Lines 2-9 computes the indices of the first length *k* sub-string and its reverse complement of the input sequence. Line 6 computes the substring index along the forward strand while line 7 computes the index of its reverse complement. This initialization is achieved in *O*(k) time. Lines 8 and 9 update the corresponding counts of the initial k-mer and its reverse compliment respectively. Lines 10-14 iteratively compute the indices of successive k-mers and their reverse complements and update their counts accordingly. The algorithm achieves a linear time performance by computing, in lines 11 and 12 respectively, the indices of every successive length *k* sub-string and its reverse complement in constant time. In order to compute the index of a sub-string in constant time the algorithm uses the fact that every successive sub-string has as its $k - 1$ higher order digits the $k - 1$ lover order digits of its immediate predecessor. Therefore, instead of computing the value of the index of the sub-string from its k digits, line 11 simply shifts by one position to the left the $k - 1$ lower order digits of its immediate predecessor (which is achieved in constant time by taking the predecessor index value modulo 4^{k-1} multiplied by 4) and adds the new digit as a lowest order digit to give the value of the current sub-string index. This is graphically illustrated in Figure 1. (c). A similar argument follows in computing the index of the reverse complement. In computing the index of the reverse compliment, line 12 simply shifts by one position to the right the $k - 1$ higher order digits of the reverse complement of the immediate predecessor (which is achieved in constant time by taking the quotient of the division of the value of the predecessor reverse compliment index by 4) and adds the new digit as a highest order digit (by multiplying it by 4^{k-1}) to give the value of the index of the reverse compliment (see Figure 1.(b) and (c)).

In the discussions that follow, we refer to nucleotide characters converted to their radix-4 digit simply as characters instead of explicitly mentioning that they are the numerical representation of the nucleotides.

The GK-mer-Count Algorithm

The GK-MER-COUNT algorithm takes as input a sequence file, f, of DNA sequence, *s*, the length *k* of the k-mer of interest, and builds a k-mer tree embodying the frequency counts of all k-mers (and their reverse compliments) in *s*. The algorithm uses a circular buffer, Gbuffer, of length k to keep track of successive k-mers along the input sequence. The GK-MER-COUNT algorithm uses the function getchar to read a character from file *f*. It also uses the functions convert to convert a nucleotide to its corresponding radix-4 digit, (A→ 0; C →1; G → 2; T → 3), and compliment to get the compliment of a converted nucleotide, (0 → 3; 1 → 2; 2 → 1; 3 → 0). Also used in the algorithm are the two functions div and mod that returns the quotient and remainder of a division operation respectively. An internal-branch-node is an array of pointers pointing to other branch nodes. An external-branch-node is an array ofpointers pointing to *leaf nodes*. Arrays in the algorithm are zero-based arrays.

```
GK-MER-COUNT (f; k)
1  for $i \leftarrow 0$ to $k - 1$
2        $c \leftarrow getchar(f)$
3        $Gbuffer[i] \leftarrow convert(c)$
4  $root \leftarrow$ new internal-branch-node[ 4 ]
5  $origin \leftarrow 0$
6  $prefix \leftarrow$ INSERT-LEAF ( INSERT-BRANCH( origin ) )
7  while not eof(f) do
8        $c \leftarrow getchar(f)$
9        ADD2COUNT( convert(c) )
```

The GK-MER-COUNT algorithm works as follows. Lines 1-3 of the algorithm populate the buffer, Gbuffer, with the first k characters of the input sequence. Line 4 creates the first internal-branch-node at depth 0 and establishes the pointer root as its handle. Line 5 sets the variable origin to *0* to indicate the current location of the k-mer in the circular buffer, Gbuffer. In line 6, the algorithm uses the two functions Insert-Branch and Insert-Leaf to add the initial k-mer to the k-mer tree. The return value from the function Insert-Leaf which is a pointer to the external branch node of the suffix sub-branch of the k-mer, is assigned to the pointer variable, *prefix*. Lines 7-9 read successive characters from the input sequence until the end of file is reached and adds the resulting k-mers to the k-mer tree via the function ADD2COUNT.

```
INSERT-BRANCH(p)
1  $x \leftarrow root$
2  for $i \leftarrow 1$ to $k - 2$
3        $r \leftarrow Gbuffer[\ p\ ]$
4        if $x[r]$ = NIL then
5        $x[r] \leftarrow$ new internal-branch-node[ 4 ]
6        $x \leftarrow x[\ r\ ]$
7        $p \leftarrow (p + 1)$ mod $k$
8  $r \leftarrow Gbuffer[\ p\ ]$
9  if $x[r]$ = NIL then
10       $x[\ r\ ] \leftarrow$ new external-branch-node[ 4 ]
11 $x \leftarrow x[\ r\ ]$
12 return $x$
```

The function INSERT- BRANCH takes the index of the start of the current k-mer on Gbuffer and adds the sub-branch representing that k-mer to the k-mer tree. It returns a pointer to the external-branch-node of the sub-branch. The algorithm starts from the first internal branch node at depth 0 by settingthe incremental pointer variable x to root. Lines 2-7 add the remaining (*k* - 2) internal branch nodes to the tree if they do not exist. Line 3 reads the character indexed by the variable p on Gbuffer. Line 4 checks to see whether an internal branch node is already present for this character. If it is a new sequence (i.e. there is no internal branch node extending from this character) an internal branch node is added for the character in line 5. In line 6, the incremental pointer *x* is advanced to the next internal branch node indexed by *r*. In line 7, the index p is incremented to the next character of the k-mer on Gbuffer. Lines 9-10 add the external branch node of the sub-branch of the k-mer. Line 9 checks to see whether this node already exists and line 10 adds it if it does not. In line 11, the pointer variable x is moved to point to the external branch node of the k-mer sub-branch and line 12 returns this pointer to the calling routine.

```
INSERT-REVERSE-COMPLIMENT-BRANCH(p)
1  x ← root
2  for i← 1 to k− 2
3          r ← compliment(Gbuffer[ p ] )
4          if x[r] = NIL then
5              x[r] ← new internal-branch-node[ 4 ]
6          x ← x[ r ]
7          p ← (p − 1) mod k
8  r ← compliment(Gbuffer[ p ] )
9  if x[r] = NIL then
10         x[ r ] ← new external-branch-node[ 4 ]
11  x ← x[ r ]
12  return x
```

The function INSERT-REVERSE-COMPLIMENT-BRANCH is similar to the function Insert-Branch except that it adds the reverse compliment sub-branch of the k-mer in *Gbuffer* to the k-mer tree.

```
INSERT-LEAF(x)
1          p ← (origin + k − 1) mod k
2          r ← Gbuffer[p]
3          x[r] ← new leaf-node
4          x[r] → extension ← NIL          ▷ Used in PGK-MER-COUNT
5          x[r] → count ← 1
6          x[r] → suffix ←INSERT-BRANCH( (origin + 1) mod k )
7          x[r] → rcomp ← INSERT-COMPLIMENT-LEAF(INSERT-REVERSE-COMPLIMENT-BRANCH((origin−1) mod k) )
8          (x[r] → rcomp ) → rcomp ← x[r]
9          return x[r] → suffix
```

The function INSERT-LEAF takes a pointer to the external branch node of the k-mer sub-branch of the k-mer in Gbuffer and adds its *leaf node* (to complete the branch). In line 1, the index p is set to point to the last character of the k-mer in Gbuffer. The implicit index representing this character giving the location on the external branch node is read in Line 2. Line 3 adds the new *leaf node*. Line 4 initializes the extension pointer of the *leaf node*. Line 5 sets the variable count of the *leaf node* to 1. Line 6 calls the function INSERT-BRANCH to add the *k*+1 *prefix* of the k-mer to the k-mer tree and sets the *suffix* pointer of the *leaf node* to its return value establishing a pointer link between the *leaf node* and the external branch node of the *prefix* sub-branch of the k-mer. Line 7 uses the functions INSERT-REVERSE-COMPLIMENT-BRANCH and INSERT-COMPLIMENT-LEAF to add the reverse compliment of the k-mer to the k-mer tree. The return value of the function INSERT-COMPLIMENT-LEAF is assigned to the pointer variable *rcomp* establishing a pointer link between the *leaf node* of the k-mer and its reverse compliment *leaf node*. Line 8 uses the newly establishedrcomppointer to access the reverse compliment leaf node toset its rcomp pointer to point to the current *leaf node*. Line 9 returns the pointer to the external branchnode of the suffix sub-branch of the k-mer, to the calling function.

```
INSERT-COMPLIMENT-LEAF(x)
1  p ← origin
2  r ← compliment(Gbuffer[p] )
3  x[r] ← new leaf-node
4  x[r] → extension ← NIL     ▷ Used in PGK-MER-COUNT
5  x[r] → count ← 1
6      x[r] → suffix ← INSERT-REVERSE-COMPLIMENT-BRANCH( (origin − 2) mod k )
7  return x[r]
```

The function INSERT-COMPLIMENT-LEAF is similar to the function Insert-Leaf except that it does not have the two corresponding lines 7 and 8 of function INSERT-LEAF.

```
ADD2COUNT(v)
1  origin ← (origin + 1) mod k
2  Gbuffer[ (origin + k − 1) mod k ] ← v
3  if prefix [v] ≠ NIL then
4          prefix [v] → count + +
5          (prefix [v] →rcomp )→count + +
6          prefix ← prefix [v] → suffix
7  else
8          prefix ← INSERT-LEAF ( prefix  )
```

The function ADD2COUNT acts as the main function that adds k-mers to the k-mer tree as new characters are read from the input sequence. In line 1, the function ADD2COUNT sets the variable origin to the index of the new k-mer in *Gbuffer*. In line 2, the character read from the input sequence is added to Gbuffer. Line 3 checks to see if there is a *leaf node* representing the read character extending from the external branch node pointed to by the *prefix* pointer (i.e. if the k-mer is already in the k-mer tree). Ifthe *leaf node* does exist, line 4 increments by one the variable *counter* which holds the frequency of the k-mer. Line 5 uses the *rcomp* pointer to access the reverse compliment of the k-mer and increment itscounter by one. Line 6 moves the *prefix* pointer to the external branch node of the *suffix* sub-branch of the k-mer which is given by the suffix pointer in the *leaf node*. If the *leaf node* does not exist, line 8 adds it to the k-mer tree.

The PGK-MER-COUNT Algorithm

The PGK-MER-COUNT algorithm takes as input a sequence file, *f,* of a DNA sequence, *s,* the sub k-mer length, k0, and k-mer length, *k,* where 1 < k_0< *k*− 1, and, the minimum desired k-mer frequency of interest, , and builds a k-mer tree embodying the frequency counts of all k-mers (and their reverse compliments) in s with a minimum k_0 on k substring abundance greater than or equal to α. The algorithm uses a circular buffer, *PGbuffer*, of length *k*.

The PGK-MER-COUNT algorithm takes as input a sequence file, *f*, of a DNA sequence, *s*, the sub k-mer length, $k_{0,}$ and k-mer length, *k*, where $1 < k_0 < k - 1$, and, the minimum desired k-mer frequency of interest, , and builds a k-mer tree embodying the frequency counts of all k-mers (and their reverse compliments) in s with a minimum k_0 on *k* substring abundance greater than or equal to *α*. The algorithm uses a circular buffer, *PGbuffer*, of length *k*.

The PGK-MER-COUNT algorithm uses the following global variables

q ▹*PGbuffer* index(position to insert next character read from file)
a ▹*PGbuffer* index
b ▹*PGbuffer* index

n ▹ counter of consecutive number of sub k-mers with sub-string abun-dance greater than or equal to *α*.
w ▹ constant number of substrings of length k_0 in string of length *k*
m ▹ pointer to a *leaf-node*
e ▹ pointer to the leaf-node for extension

```
PGK-MER-COUNT (f; k_0; k; α )
1     for i← 0 to k_0– 1
2     c ← getchar(f)
3     Gbuffer[i] ← convert(c)
4     root ← new internal-branch-node[ 4 ]
5     origin ← 0
6     prefix ← INSERT-LEAF ( INSERT-BRANCH(( origin ) )
7     while not eof(f) do
8     c ← getchar(f)
9     ADD2COUNT( convert(c) )
10    move file pointer to beginning of file (f)
11    for i← 0 to k– 1
12    c ← getchar(f)
13    PGbuffer[i] ← convert(c)
14    w ← k – k_0+ 1
15    a ← k_0 – 1
16    b ← 0
17    q ← k – 1
18    l ← root
19    while b < (k_0– 1) do
20    l ← l[PGbuffer[b]]
21    b + +
22    m ← l[PGbuffer[b]]
23    e ← m
24    n ← 0
25    PG-Do()
26    while not eof(f) do
27    c ← getchar(f)
28    PG-ADD2COUNT( convert(c) )
```

The PGK-MER-COUNT algorithm performs two scans of the input sequence. The first scan given in lines 1-9 implements the GK-MER-COUNT algorithm described above with a sub k-mer length k_0 to generate a k_0-mer tree. This k_0-mer tree is built upon by the second scan of the algorithm by adding branch extensions for k-mers with a minimum k_0 on *k* substring abundance greater than or equal to *α* found in the input sequence. Line 10 resets the file pointer to the beginning of the input file. Lines 11-13 populates the circular buffer, *PGbuffer*, with the first *k* characters of the input sequence. In line 14, the constant *w* is assigned the length of the branch extension. In line 15, the index variable a is set to the index of the last character of the first k_0-mer of the k-mer in *PGbuffer*. In line 16, the index variable b is set to the first character of the k-mer in *PGbuffer* and in line 17, the index variable *q* is set to thelast character of the k-mer in *PGbuffer*. In line 18, the incremental pointer *l* is set to point to the first internal branch node of thek_0-mer tree. Lines 19-21 move the incremental pointer l down the tree fromthe first internal branch node to the external branch node of the first k_0-mer of the k-mer in *PGbuffer* by traversing *PGbuffer* using the index variable b. In line 22, the pointer variable m is set to point to the *leaf node* of the k_0-mer. In line 23, the pointer variable e is also set to point to the *leaf node* of the k_0-mer. The pointer variable *e* is used to access the *leaf node* of the k_0-mer when a branch extension is added. In line 24, the counter variable *n* is set to zero. The variable n is used to count the number of k_0-mers in the k-mer with a frequency greater than . Line 25 calles the function PG-Do which decides whether a branch extension should be added to the k_0-mer tree. Lines 26-28 read successive characters from the input sequence until the end of file is reached and adds the resulting k-mers to the k_0-mer tree if their minimum k_0 on *k* substring abundance is greater than or equal to *α*, via the function PG-ADD2COUNT.

```
PG-Do()
1     if m→count≥ α then
2     n + +
3     while m→count≥ α and n < w do
4     b ← (b + 1) mod k
5     m ← m → suffix [PGbuffer[b]]
6     if m→count≥  then
7     n + +
8     if n = w then
9     PG-ADD-EXTENSION()
```

The purpose of the function PG-Do is to add a branch extension to the k_0-mer tree to represent the k-mer in *PGbuffer* if its minimum k_0 on *k* substring abundance is greater than or equal to *α*. At the beginning the pointer variable m points the *leaf node* of the first k_0-mer of the k-mer. Line 1 of function PG-Do checks to see whether this k_0-mer has a frequency greater than or equal to *α*. If it does, line 2 increments the counter variable n by one. Lines 3-7 scan the k-mer until a k_0-mer with a frequency less than *α* is encountered or until all k_0-mers of the k-mer are accounted for. Line 4 increments the index variable b which keeps track of the last character of the k_0-mer. Line 5 uses the suffix pointer of the *leaf node* of the current k_0-mer and the index to the *leaf node* of the next k_0-mer to move the pointervariable m to the *leaf node* of the next k_0-mer. Line 6 checks to see if the new k_0-mer has a frequency greater than or equal to α and if it does line 7 increments the counter variable n. The condition n = w in line 8 is satisfied if and only if the k-mer has a minimum k_0 on k substring abundance greater than orequal to α. Line 9 calls the function PG-Add-Extension to add a branch extension to the k_0-mer tree to represent the k-mer if the condition in line 8 is satisfied.

```
PG-ADD-EXTENSION()
1      if e→extension = NIL then
2 e → extension ← new internal-branch-node[ 4 ]
3 x ← e → extension
4      p ← (a + 1) mod k
5      for i← 2 to k−$k_0$− 1
6      r ← PGbuffer[p]]
7      if x[r] = NIL then
8      x[r] ← new internal-branch-node[ 4 ]
9      x ← x[r]
10     p ← (p + 1) mod k
11     r ← PGbuffer[p]]
12     if x[r] = NIL then
13     x[r] ← new external-branch-node[ 4 ]
14     x ← x[r]
15     p ← (p + 1) mod k
16     r ← PGbuffer[p]]
17     if x[r] = NIL then
18     x[r] ← new leaf-node
19     x[r] → extension ← NIL
20     x[r] → count ← 0
21     x[r] → suffix ← NIL
22     x[r] → rcomp ← PG-ADD-EXTENSION-REVERSE-COMPLIMENT()
23     (x[r] →rcomp ) →rcomp←x[r]
24     x[r] → count + +
25     (x[r] →rcomp ) →count + +
```

The function PG-ADD-EXTENSION adds a branch extension to the k_0-mer tree for a new k-mer or increments the frequency count of a k-mer already having a branch extension in the k_0-mer tree. The branch extension is added at the *leaf node* of the first k_0-mer of the k-mer. Access to this *leaf node* isgained by the pointer variable *e*. In line 1, the function checks to see whether there is *a* branch extension from the *leaf node*. If the *leaf node* has no branch extension, line 2 adds the first internal branch node of the extension. In line 3, the incremental pointer variable *x* is pointed to the first internal branch node of the extended branch node of the k-mer. The index variable a holds the index of the last character of the first k_0-mer of the k-mer on *PGbuffer*. In line 4, the index variable *p* is assigned the next index from a which is the index of the first character of the k-mer in its branch extension. Lines 5-10 add new internal branch nodes to the extending branch if they do not exist. Line 6 gets the next character from *PGbuffer*. Line 7 checks to see if an internal branch node already exists for this character. Line 8 adds anew internal branch node to the growing branch if one does not already exist. Line 9 moves the pointer variable *x* to the node pointed to by the character indexed by *r* in the internal branch node. Line 10 moves the index variable *p* to the next character on *PGbuffer*. Line 11 gets the character pointed to by the index variable p. Line 12 checks to see if an external branch node already exists for this character. If one does not exist, line 13 adds a new external branch node. Line 14 moves the pointer variable *x* to the external branch node pointed to by index r of the internal branch node. Line 15 moves the index variable *p* to the next character on *PGbuffer* which must be the last character of the k-mer. Line 16 getsthis character from *PGbuffer*. Line 17 checks to see if a *leaf node* exists in the extended branch for the character. If one does not exist lines 18-23 add a *leaf node* and sets its variables to their initial values. Line 24 increments the frequency count of the k-mer and line 25 increments the frequency count of its reverse compliment.

```
PG-ADD-EXTENSION-REVERSE-COMPLIMEN()
1      x ← root
2      p ← (a + k − $k_0$) mod k
3      for i← 1 to $k_0$− 1
4      r ← compliment(PGbuffer[p])
5      x ← x[r]
6      p ← (p − 1) mod k
7      r ← compliment(PGbuffer[p])
8      if x[r] →extension = NIL then
9      x[r] → extension ← new internal-branch-node[ 4 ]
10     x ← x[r] → extension
11     p ← (p − 1) mod k
12     for i← 2 to k−$k_0$− 1
13     r ← compliment(PGbuffer[p])
14     if x[r] = NIL then
15     x[r] ← new internal-branch-node[ 4 ]
16     x ← x[r]
17     p ← (p − 1) mod k
18     r ← compliment(PGbuffer[p])
19     if x[r] = NIL then
20     x[r] ← new external-branch-node[ 4 ]
21     x ← x[r]
22     p ← (p − 1) mod k
23     r ← compliment(PGbuffer[p])
24     if x[r] = NIL then
25     x[r] ← new leaf-node
26     x[r] → extension ← NIL
27     x[r] → count ← 0
28     x[r] → suffix ← NIL
29     x[r] → rcomp ← NIL
30     return x[r]
```

The function PG-Add-Extension-Reverse-Compliment adds the reverse compliment of a k-mer extension to the k_0-mer tree. It is similar to the function PG-Add-Extension, however it needs to trace down the k_0-mer tree using the first k_0 characters of the reverse compliment to locate the correct *leaf node* to add the reverse compliment extension. This process is carried out in lines 1-6. The function returns a pointer to the *leaf node* of the extension.

PG-ADD2COUNT(v)

```
1    $q \leftarrow (q + 1)$ mod k
2    $PGbuffer[q] \leftarrow v$
3    if $n = w$ then
4    $a \leftarrow (a + 1)$ mod $k$
5    $e \leftarrow e \rightarrow$ suffix $[PGbuffer[a]]$
6    $b \leftarrow (b + 1)$ mod $k$
7    $m \leftarrow m \rightarrow$ suffix $[PGbuffer[b]]$
8    $n - -$
9    PG-Do()
10   else if $a = b$ then
11   $a \leftarrow (a + 1)$ mod $k$
12   $e \leftarrow e \rightarrow$ suffix $[PGbuffer[a]]$
13   $b \leftarrow (b + 1)$ mod k
14   $m \leftarrow m \rightarrow$ suffix $[PGbuffer[b]]$
15   PG-Do()
16   else
17   $a \leftarrow (a + 1)$ mod $k$
18   $e \leftarrow e \rightarrow$ suffix $[PGbuffer[a]]$
19   $n - -$
```

The function PG-ADD2COUNT is the main function that adds branch extensions to the k_0-mer tree for qualifying k-mers (i.e. k-mers with a minimum k_0 on k substring abundance greater than or equal to α) as new characters are read from the input sequence. Line 1 of PG-ADD2COUNT moves the index variable q by one to the point on *PGbuffer* where the new character, v, is inserted. Line 2 adds this character to *PGbuffer*. In line 3, the algorithm checks to see if the most recent k-mer was a qualifying k-mer. If this is the case, the algorithm only needs to check if the last k_0-mer of the current k-mer has a frequency greater than or equal to α to be a qualifying k-mer. A qualifying k-mer has w k_0-mers with frequency greater than or equal to α . The number of k_0-mers in a k-mer with a frequency greater than or equal to α is recorded in the variable n. When the index variables a and b are equal fulfilling the condition in line 10, it implies that $n = 0$ or that counting frequencies of k_0-mers in the k-mer must be done from the first k_0-mer, otherwise, when $a \neq b$ the algorithm executes lines 17-19 which in effect discards the next $n(< w)$ k-mers which are guaranteed to have a minimum k_0 on k substring abundance less than α.

References

1) Liolios K, Mavromatis K, Tavernarakis N, Kyrpides NC (2008) The Genomes On Line Database (GOLD) in 2007: status of genomic and metagenomic projects and their associated metadata. Nucleic Acids Res 36: 475–479.

2) Vincens P, Buffat L, André C, Chevrolat JP, Boisvieux JF, et al. (1998) A strategy for finding regions of similarity in complete genome sequences. Bioinformatics 14: 715–725.

3) Kurtz S, Schleiermacher C (1999) REPuter: Fast computation of maximal repeats in complete genomes. Bioinformatics 15(5): 426–427.

4) Paolo Ferragina , Roberto Grossi (1999) The string B-tree: A new data structure for string search in external memory and its applications. J Assoc Comput Mach 46: 236–280.

5) Choi JH, G CH (2002) Analysis of common k-mers for whole genome sequences using SSB-tree. Genome Inform 13: 30–41.

6) Marais G, Kingsford C (2011) A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics 27(6): 764–770.

7) Karp RM, Rabin MO (1987) Efficient randomized pattern-matching algorithms. IBM J Res Dev 31(2): 249–260.

8) Hubbard TJP, Aken BL, Beal K, Ballester B, Caccamo M, et al. (2007) Ensembl 2007. Nucleic Acids Res (Database issue) 35: 610–617.

9) Stoesser G, Baker W, van den Broek AE, Camon E, Hingamp P, et al. (2000) The EMBL Nucleotide Sequence Database. Nucleic Acids Res 28: 19–23.

10) Khil PP, Camerini-Otero RD (2005) Molecular Features and Functional Constraints in the Evolu-tion of the Mammalian X Chromosome. Critical Reviews in Biochemistry and Molecular Biology 40(6): 313–330.