SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Analysing k-mer distributions in a genome sequencing project

av

Josefine Röhss

2014 - No 22

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Analysing k-mer distributions in a genome sequencing project

Josefine Röhss

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Kristoffer Sahlin

2014

Thesis
Analysing k-mer distributions in a genome sequencing project

Josefine Rohss

September 2, 2014

Abstract

The usage of next generation sequencing for de novo assembly of genomes is increasing rapidly. However, due to the short read length from the next generation sequencing protocols, the assembly process is complicated. Algorithms for genome assembly need to be developed further in order to obtain high quality results that meet the criteria for downstream analysis. One subject for improvement is choosing an optimal k-mer size, depending on certain other variables. This project examines how different genome and sequencing conditions such as the number of repeats in the genome, GC-content and coverage can affect the choice of k-mer size. We tested KmerGenie, a program designed to calculate the best value of k. We also developed programs that simulated a genome from a Markov chain and divided the genome into reads with which KmerGenie predicted the best value of k and developed our own program to divide the reads into k-mers and produce histograms that could be compared to the output from KmerGenie. We tested different values of coverage, GC-content and repeat content and all the outputs from KmerGenie were compared. The result from the tests show that KmerGenie is not always able to predict the best value of k. Depending on repeats in the genome and GC-content, the quality of the estimation can vary substantially.

Contents

1 Definitions

Genome
The complete set of genetic information in an organism. It contains all the information that the organism requires to function. In living organisms, the genome is stored in the organism's DNA[1].

Read
Fragments created from a genome that can be sequenced and reassembled.[2]

Paired end read
Reads created by sequencing a fragment of the genome from both ends.[2].

Substitution
A point mutation where a single nucleotide is substituted with a different nucleotide[3].

Tandem duplication
A DNA sequence that is copied and inserted next to the original sequence[3].

Interspread duplication
A DNA sequence that is copied and inserted in a different place in the genome than the original sequence. The two copies can be separated by one up to millions of nucleotides[3].

Insertion
A single or several nucleotides are inserted somewhere in the genome[3].

Inversion
The reversed complement of a sequence is inserted somewhere in the genome[3].

K-mers
Sequences that are created when reads are broken up into all possible parts of a fixed length k[13].

Contigs
A set of overlapping DNA segments from a single genome, from which the complete sequence may be deduced.[2]

Read error
Base pairs that are sequenced when creating reads can be substituted to some other base pair because of read errors, leading to an incorrect read.

Coverage
The average number of reads that align to each base within the DNA[2].

GC-content
The percentage of guanine and cytosine in a sequence of base pairs, usually expressed in percentage of total bases.

De novo sequencing
Sequencing short reads to create a full-length sequence from an organism without a known genome sequence, without a reference genome[2].

De Bruijn graph
A directed graph representing overlaps between sequences of symbols, in this case k-mers.[10]

Markov model
A stochastic model that is built of a Markov chain[5].

Markov chain
Let $[X_n, n = 0, 1, 2...]$ be a stochastic process that takes on a finite or countable infinite number of possible values. If $X_n = i$ the process is said to be in state i at time n. We suppose that whenever the process is in state i, there is a fixed probability P_{ij} that it will next be in state j. That is, we suppose that

$$P\{X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, ..., X_1 = i_1, X_0 = i_0\} = P_{ij}$$

For all states $i_0, i_1, ..., i_{n-1}, i, j$ and all $n >= 0$. Such a stochastic process is known as a Markov chain.
The possible values of X_n form a set that is called the state space of the chain.[4][5]

2 Introduction

Since the human genome was fully assembled in 2003 genome sequencing protocols have greatly improved and genome assembly has become popular due to low cost per base.[6]. The techniques are continuously improving, but since genome sequencing and assembly programs are still developing fields, new algorithms and improvements of the programs are still needed in order to obtain assemblies of higher quality. Several factors influence the ease or difficulty with which a genome can be sequenced and assembled. Four of those factors are the GC-content in the reads, the number of repeats in the genome, the coverage and the k-mer size. The GC-content affects the sequencing by creating more uneven coverage in some regions while repeats, read errors and coverage can affect the quality of the assembly. The k-mer size also affects the assembly, but this is something we can control and vary within the assembler. Finding optimal values for these parameters is an important task to improve the assembly process[13]. Choosing a good k-mer size is important to obtain long and correct contigs. Too short k-mer sizes cannot span over smaller repeats, but too large k-mer sizes will make the de Bruijn graph separated (because of insufficient overlaps of reads). A good choice of k will make a de Bruijn graph-based assembler produce long and accurate contigs which in turn enables tools for ordering and orientation of contigs to create an accurate lay out of the contigs.

In this project, genomes have been simulated in two different ways. Some genomes were created with moderate repeat content and some genomes were created with a higher repeat content, because of the difficulty that too many repeats presents to sequence alignments and assembly programs[7]. The aim was to make this process similar to the evolution of the DNA. With these genomes, different combinations of GC-contents, coverages and lengths of k-mers were tested to determine when the best result was reached. The simulation and testing was implemented in Python programs that models the creation of the genome and the partition of the genome into reads. These reads were then used by another program to simulate k-mers and create histograms of the result. The same reads were used by the program KmerGenie to predict the best value of k and create histograms that were compared to the histograms already created.

The mathematical approach used to create the genome as similar to the evolution as possible was Markov modeling and Markov chains. Hence, one part of this thesis discusses those topics.

3 Background

3.1 The Human Genome

A genome is the genetic information of an organism, encoded in DNA (deoxyribonucleic acid). The DNA consists of two strands, wrapped around each other in the form of a double helix. The strands are arranged with the bases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T) next to each other which together (A with T and C with G) form pairs between the two strands[1].

Evolution of today's genomes began with just a few base pairs and since then it has evolved with time to include approximately three billion base pairs today. The reason for the increasing length is different kinds of mutations such as substitutions, tandem duplications, interspersed duplications, insertions and inversions.[8]

Due to this evolutionary process, some parts of the human genome consist of repetitive DNA. Repetitive DNA is a sequence of base pairs that occurs in multiple places of the genome, either next to each other or in different places of the genome. The length of the repetitive sequence can be from one base pair up to millions of base pairs. The human DNA consists of approximately 50 percent repetitive sequences of different lengths[7]. If a genomic sequence is to be regarded as a repeat or not depends on what sequence length we are considering. If we consider a genomic sequence of length x, a repeat is a sequence of length y where $y > x$ that occurs more than one time in the genome.

3.2 Next generation sequencing

Next generation sequencing is a new technological breakthrough that has taken place in the last few years. It is used to sequence large numbers of DNA fragments and produce millions of reads every run. The major advantage offered by next generation sequencing is the ability to produce large volumes of data at a comparatively low cost. The two most important applications for next generation sequencing is resequencing of the human genome to improve our understanding of how genetic differences affect human health and to sequence entire genomes of several related organisms to compare them and be able to study the evolution of the organisms[9].

The Human Genome Project started in 1990 and was finished in 2003. The project identified many genes and determined the sequences of most of the base pairs in the human DNA to create a reference DNA sequence for the human genome[6]. After the sequencing of the human genome was completed, the efficiency of sequencing has increased continuously and more and more large genomes have been assembled. At the time of the Human Genome Project it took 10 years to sequence the entire human genome, but next generation sequencing machines now accomplish that task in only a few hours.

Next generation sequencing and assembly begins with constructing reads from the genome. Reads are created by dividing the genome into fragments of the desired length of the reads. After that, the reads are divided into k-mers which are all possible fragments of the read with length k. The assembling program then uses all these k-mers to reconstruct the genome with as few errors as possible. Many of the assemblers are based on de Bruijn graph framework where the assembler constructs graphs, performs graph simplifications, and outputs non-branching paths as contigs which the assembler predicts are in the genome[13]. The errors that can occur in this process can for example be incomplete and fragmentally assembled sequences due to too much repetitive DNA. This type of error mainly occurs when there are many repeats larger than the k-mer length.

3.3 Markov chains

A Markov chain is a stochastic model within the Markov models, which transitions from one state to another where the next state only depends on the current state.

A Markov chain is a mathematical system which is assumed to be in any of a finite (or countable infinite) number of states. Every state represents a particular condition of the system and can be determined from the current state, without knowledge of the states in the past. The changes of states are called transitions and a Markov chain with a limited number of states can be represented by a transition matrix with all the probabilities for every possible change of state. A Markov chain can be used for describing systems that follow a chain of linked events, where the next event depends only on the current state of the system[5].

3.4 Genome assembly

Genome assembly is the process of finding overlaps between reads, merging the overlapping reads together and creating a consensus string that hopefully resembles the genome that was sequenced.
A common method for finding the overlap between reads is to create a de Bruijn graph of k-mers from the reads. A De Bruijn graph is a directed graph with d^n nodes labeled by n-tuples over a d-character alphabet. The edges are defined to be ordered pairs of the form $((\alpha_1...\alpha_n),(\alpha_2...\alpha_n\alpha_{n+1}))$ where α_{n+1} is any character in the alphabet[10].

In an assembly the alphabet consists of four letters (A, T, C, G). There are two strategies for genome assembly, it can either be constructed by a Hamiltonian cycle or an Eulerian cycle[11].
A Hamiltonian cycle of a graph G is a cycle of G which visits every node exactly once. An Eulerian cycle of G is a cycle of G which traverses every edge exactly once[10].
In a Hamilton cycle the vertices in the graph is k-mers and the edges are pairwise alignments. Walking along a Hamiltonian cycle allows one to reconstruct the genome by forming an alignment in which each successive k-mer is shifted by one position. This process recovers the genome but does not scale well to large graphs.
In an Eulerian cycle the vertices are (k-1)-mers and the edges are k-mers. This is the technique used in modern short-read assembly. Finding an Eulerian cycle allows one to reconstruct the genome by forming an alignment in which each successive k-mer is shifted by one position. This generates the same genome sequence without performing the computationally expensive task of finding a Hamiltonian cycle. Therefore an Eulerian cycle is easier to solve than a Hamilton cycle[11].

3.5 Choosing optimal k-mer size

Choosing an optimal k-mer size is essential for the quality of the results from the de Bruijn graph-based assembler. K-mers should be long enough to span

over repeats of length k-1. However, k-mers should be small enough to be able to create a de Bruijn graph that is sufficiently connected to form long contigs. To obtain an accurately reconstructed genome, the choice of k-mer size is of vital importance.

There are other already existing models and programs that can be useful tools for sequencing and assembly. They can for example help with calculations of the length of the genome, the length of the k-mers or the k-mer length that is most favorable to use for the genome to be assembled. KmerGenie is a program that, given a set of reads, calculates a suggestion for the best length of the k-mers. It takes a file with reads as input and gives a sample report with all the results, including a plot for the best value of k and histograms for each value of k as output. When this program first was published it provided valuable assistance for de novo sequencing, and experiments have shown that KmerGenie's choices lead to assemblies that are close to optimal[13].

4 Methods

To be able to examine the different parameters, the programing language Python was used to create several different programs. The method can be summarized as first simulating six genomes with a varying number of repeats and different content of GC. Reads were simulated from these genomes and were used by the program KmerGenie to be able to compare the results for different parameters of coverage, repeats and GC-content. K-mers of different lengths were simulated from the reads to create histograms to be compared to the histograms from KmerGenie. All the programs are described below and the code and the plots can be found in the Appendix of the thesis.

4.1 The genomes

The aim was to create six different genomes with different amounts of repeats and GC-content. The reason for this is that a sequence with more repeats or a higher content of the nucleotides G and C is much more complicated to sequence and assembly and the sequencing more likely to result in errors[7][12]. The reason for testing different genomes is to see what difference repeats and GC-content makes to the best choice of k-mer length.

We started with a genome of 500 base pairs and simulated evolution of the genome with substitutions, tandem duplications, interspersed duplications, insertions and inversions. The initial 500 base pairs were created in different ways depending on how much GC-content that was wanted in the final genome. To obtain a normal amount (about 50%) of G and C the first base pairs were created randomly. To obtain a lower content the starting sequence was created with 30% G and C, and to obtain a higher content it was created with 60% G and C.

The other task was to simulate genomes with different amounts of repeats. This was done by simulating evolution of the human genome with different types of mutations. To obtain a normal rate of repeats, most of the mutations (80%) were substitutions since this mutation only substitutes one nucleotide with another and hence does not cause any repeats. To obtain a higher amount of repeats the rate of substitution was much lower (10%) and instead there were more duplication mutations that result in repeats since that type of mutation copies a part of the sequence and inserts it somewhere in the genome.

Using this method, six different genomes of 1 000 000 base pairs were created These genomes are described in Table 1.

Table 1: The genomes

Genome	Amount of repeats	GC-content (%)
1	normal	50
2	higher	50
3	normal	30
4	higher	30
5	normal	60
6	higher	60

This table shows the GC-content and amount of repeats for all of the six genomes that were created.

4.2 Markov chain calculations

In the program creating the genomes described above, a Markov chain was used to simulate evolution with the different mutations. The reason for using Markov chains in this program is that the mutations are not equally common and depend on the previous mutation. With the transition matrices used in Markov chains, it is easy to change the transition probabilities to obtain the desired results. In this case the transition probability for substitutions can be changed to obtain a genome with different amounts of repeats.

To accomplish this, five different mutations were used as states in the Markov chain. Given the current state, the states represent whether the next state will be a substitution (Sub), a tandem duplication ($TDup$), an interspersed duplication ($IDup$), an insertion (Ins) or an inversion (Inv). Labeling the state space $[1 = Sub, 2 = TDup, 3 = IDup, 4 = Ins, 5 = Inv]$ we get the two different transition matrices P_1 and P_2 below.

$$P_1 = \begin{pmatrix} 0.8 & 0.08 & 0.02 & 0.05 & 0.05 \\ 0.8 & 0.06 & 0.03 & 0.055 & 0.055 \\ 0.8 & 0.085 & 0.01 & 0.0525 & 0.0525 \\ 0.8 & 0.085 & 0.025 & 0.03 & 0.06 \\ 0.8 & 0.085 & 0.025 & 0.06 & 0.03 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 0.1 & 0.4 & 0.2 & 0.1 & 0.2 \\ 0.1 & 0.25 & 0.25 & 0.15 & 0.25 \\ 0.1 & 0.45 & 0.1 & 0.1 & 0.25 \\ 0.1 & 0.425 & 0.225 & 0.05 & 0.2 \\ 0.1 & 0.45 & 0.25 & 0.1 & 0.1 \end{pmatrix}$$

The difference between the two transition matrices is that P_2 has a lower transition probability for substitutions and a higher transition probability for the other mutations, especially tandem duplications and interspersed duplications, than P_1. These transition probabilities will result in more repeats in the genome created with P_2. The reason for the two different transition matrices is that we want to compare the result of a genome with more repeats to the result of a genome with fewer repeats.

Calculation of the steady-state probabilities for the transition matrices gives the final rate of all the different mutations. This demonstrates what percentage share every specific mutation can be presumed to comprise of the total set of mutations that were used to obtain the final genomes. Depending on the steady-state probability for substitution, it is also possible to predict the amount of repeats we can expect in the genomes. Calculation of the steady-state probabilities are executed below.

$$\lim_{N\to\infty} P_1^N = \begin{pmatrix} 0.8 & 0.0790244 & 0.0210789 & 0.0499483 & 0.0499483 \\ 0.8 & 0.0790244 & 0.0210789 & 0.0499483 & 0.0499483 \\ 0.8 & 0.0790244 & 0.0210789 & 0.0499483 & 0.0499483 \\ 0.8 & 0.0790244 & 0.0210789 & 0.0499483 & 0.0499483 \\ 0.8 & 0.0790244 & 0.0210789 & 0.0499483 & 0.0499483 \end{pmatrix}$$

These steady-state probabilities indicate that of all the mutations that will occur during the simulation of the genome, 80% of the mutations will be substitutions, 7.9% will be tandem duplications, 2.1% will be interspersed duplications, 5.0% will be insertions and 5.0% will be inversions. The high percentage of substitutions indicates that the genomes created with this transition matrix will have a low amount of repeats.

If we calculate the eigenvalues for the same matrix we get one value very close to 0 and one very close to 1. This may mean that numerical calculations from the program in this case not are reliable.

$$\lim_{N\to\infty} P_2^N = \begin{pmatrix} 0.1 & 0.368484 & 0.210592 & 0.112785 & 0.20814 \\ 0.1 & 0.368484 & 0.210592 & 0.112785 & 0.20814 \\ 0.1 & 0.368484 & 0.210592 & 0.112785 & 0.20814 \\ 0.1 & 0.368484 & 0.210592 & 0.112785 & 0.20814 \\ 0.1 & 0.368484 & 0.210592 & 0.112785 & 0.20814 \end{pmatrix}$$

In this case the steady-state probabilities indicate that 10% of the mutations will be substitutions, 36.8% will be tandem duplications, 21.1% will be interspersed duplications, 11.3% will be insertions and 20.8% will be inversions. The low percentage of substitutions indicates that the genomes created with this transition matrix will have a high amount of repeats.

If we calculate the eigenvalues for this matrix we also get one value very close to 0 and one very close to 1. This may mean even in this case that the numerical calculations from the program not are reliable.

This will result in genomes with different content of repeats depending on which transition matrix that was used to create the genome. Since we want Genome1, Genome3 and Genome5 to have a normal amount of repeats, transition matrix P_1 will be used in the Markov chain to create these genomes. Genome2, Genome4 and Genome6 are supposed to have a higher amount of repeats and therefore transition matrix P_2 will be used in the Markov chain in the simulation of these genomes.

The length of the tandem duplications, interspersed duplications, insertions and inversions were decided by different distributions. The length of the duplications and inversions were decided with a normal distribution with the mean 2 and the standard deviation $\frac{G}{20}$ (G = length of the genome) and the insertions were decided with an exponential distribution with mean $\frac{1}{10}$. These distributions are estimated and do not have to be the exact same as in evolution.

4.3 Reads

When the six different genomes had been simulated, the next step was to divide them into paired end reads[2] with a certain length and simulate the sequencing of these reads with respect to read errors and GC-content. This was accomplished with another program in Python. In this experiment the read length 150 base pairs was chosen and the read error rate was one error approximately every 200th base pair.

The simulation of the reads in this program depends on two parameters, read errors and GC-content. The rate of read errors was approximately one every 200 base pairs which means that every 200th base pair in all of the reads was exchanged for another base pair. With more G and C in the read, the read becomes more difficult to sequence, which causes some of the reads not to be sequenced. In a paper by Michael G Ross entitled *Characterizing and measuring bias in sequence data*[12], the correlation between GC-content and the difficulty

of sequencing reads is examined. We used the relation from figure 2, E. Coli, in the paper by Ross to simulate reads as follows. The sequencing probability for a read with given GC-content that we find in Ross' figure is that if there is less than 20% G and C, there is a 100% chance that the read is sequenced. If there is more than 80% G and C, there is a 58% chance that the read is sequenced since $\frac{0.75}{1.3} * 100 = 58$. The percentage in between is described by a linear equation $y = -0.7x + 114$. Therefore, the following model was used to determine whether a read would be sequenced or not depending on the GC-content.

Table 2: Sequencing depending on GC

GC-content(%)	probability (%)
0-19	100
20-79	$-0.7(GC - content) + 114$
80-100	58

This table shows how likely it is for a read to be sequenced depending on the GC-content in the read.

The simulation of reads was performed with two different coverages, 50 and 100 respectively. With different coverages we aim to test if KmerGenie predict a better value of k depending on the coverage and the different parameters that are tested. Higher coverage should result in a larger predicted k from KmerGenie than a lower coverage should.

With this model, reads from all the six genomes were sequenced with paired end sequencing that sequence a read from each end of a fragment from the genome. This is used to decrease the errors and gaps in the final result despite repeats.

4.4 K-mers

When all the reads were sequenced, they were saved as a multiset of reads that could be divided into k-mers of a specific length. If the length of the k-mers was decided to be k, the first k-mer was created from the k first nucleotides of the read. The next k-mer was created from the nucleotides of number $2, 3, ..., k + 1$ in the read. This continued until the last k-mer from that specific read was created of the nucleotides of number $150 - k + 1, 150 - k, ..., 150$ in the read (since the length of the reads was 150). For example if we have the read ATAGATA and $k = 3$, the set of 3-mers will be ATA, TAG, AGA, GAT, ATA. This was executed three times for every set of reads with 30-mers, 40-mers and 50-mers.

When all the reads were divided into k-mers of the chosen length, the result was plotted in a histogram with the k-mer abundance (the number of times the k-mer appears in the multiset of k-mers) on the x-axis and the abundance frequency (the number of k-mers with the same abundance) on the y-axis. These histograms were created to be able to compare them to the histograms from KmerGenie. The histograms can be found in the Appendix.

4.5 Repetitive use of KmerGenie

The program KmerGenie takes one file containing reads from both ends as input and predicts the best value of k. The output is an html report of all the results and histograms for each value of k.

KmerGenie constructs approximate abundance histograms for possible values of k. The abundance histograms show the distribution of k-mer abundances (the number of occurrences in the data) for a single k value. It also contributes with a fast method for choosing the best possible value of k, based on the generated abundance histograms for the different values of k. The method for choosing k is based on the intuition that the best choice of k is the one which provides the most distinct non-erroneous k-mers to the assembler. The k-mer abundance histogram is a mixture of genomic k-mers (k-mers without errors), and erroneous k-mers. The approach of KmerGenie is to take a generative model, fit it to the histogram and derive the number of genomic k-mers of the model.

The reads created as explained above were used as input to KmerGenie to predict the best value of k for that particular set of reads. In this way the effect of repeats, GC-content and chosen coverage could be seen in the different values of k that KmerGenie predicted to be the best, and in how easy or difficult it was for KmerGenie to predict the best value, which could be concluded from looking at the histograms.

The expected result from KmerGenie is a larger k value for $coverage = 100$ than for $coverage = 50$, a higher k value for the genomes with more repeats (where transition matrix P_2 was used) than with less repeats and that it was easier for KmerGenie to predict the best value of k for the genomes with less repeats. The reason for this is that too many repeats longer than k nucleotides can be a cause of gaps in the result, so the k-mer should be longer in genomes with more repeats than in others (but not too long since a long k-mer is more likely to have an error in it)[13].

5 Results and Discussion

The lengths of the six genomes that were created were in the interval of $1010024-1042426$ (Genome1 = 1028986, Genome2 = 1042426, Genome3 = 1022043, Genome4 = 1039186, Genome5 = 1037941, Genome6 = 1010024) base pairs long. Thus they were not exactly identical in length, but the variation is considered to be sufficiently small to be negligible. The percentage of GC-content also varied slightly (Genome1 = $50,2\%$, Genome2 = $50,4\%$, Genome3 = $29,9\%$, Genome4 = $29,2\%$, Genome5 = $61,4\%$, Genome6 = $61,6\%$), but the percentages were close enough to the desired percentage that the differences can be ignored. The amount of repeats in the genome varied as desired, with more repeats in Genome2, Genome4 and Genome6 than in the rest of the genomes. It was approximately the doubled amount of repeats in the genomes created with transition matrix P_2 than in genomes created with P_1

The program KmerGenie was run with the reads created from the different genomes and different parameters with the Python programs. KmerGenie tested values of k between 21 and 121. All the predicted values of k were between 31 and 53 (see the exact values in Table 3) with the largest values belonging to Genome2 and Genome6, but overall the values lie within a narrow range.

Table 3: Result KmerGenie

Coverage	Genome1	Genome2	Genome3	Genome4	Genome5	Genome6
50	27	45	33	37	43	41
100	37	47	37	47	43	49

This table shows the predicted best value of k from KmerGenie depending on which genome the reads were created from and the coverage. The genomes that were used are written at the top of the table and the coverage is written on the left.

The graphs created from our tests and from KmerGenie for all the predicted best values are to be found in the Appendix.

In Table 3 we can see that, as we expected, there is a difference between all of the best values of k depending on the coverage, repeats and GC-content.

We predicted that a high coverage would result in a larger predicted value of k from KmerGenie than a low coverage would, and we can see in the results that this is true. The predicted k from the sequencing with coverage 50 is smaller than the k from the sequencing with coverage 100 in every case except for Genome5 where it is the same value.
We also expected a larger value of k from the reads of the genomes in which transition matrix P_2 was used (Genome2, Genome4, Genome6) since these genomes have more repeats than the rest. In the results we can see that this is true in almost every case if the results from these genomes are compared to the genome with the same GC-content but that was simulated with transition matrix P_1. It is only if we compare Genome5 with Genome6 when coverage 50 was used that this is not true. It can be several reasons for this, for example too many repeats or too big difference in length of the genome (it is a bigger difference in length between these genomes than in the other cases).
The expected results from KmerGenie with respect to the different GC-contents was not as clear as the other predictions, and we can see in the results in Table 3 that it is difficult to find a connection between the predicted values of k and the GC-content in the genomes. We can see that almost all of the values of k from the genomes with 70% GC are higher than the other and almost all of the values of k from genomes with 30% GC are lower than the other values, but it is not a connection distinct enough to make the conclusion that this is explicitly due to the GC-content. To get a clearer result depending on GC-content one more test should be executed with GC-content as the only parameter to be tested.

In some of the graphs from KmerGenie there is no clear global maximum but instead multiple local maxima (we can for example see this in Figure 15). These cases reflect that the statistical model in KmerGenie does not always correctly

fit the input data for some values of k or that the genome contains many repeats of a small length, and the predicted k may not be the best. In these cases, KmerGenie recommends to try a larger k if there is a maximum at a larger k than the one suggested. Especially in the graphs from the genomes which were simulated with transition matrix P_2 we can see multiple maxima and it may be a different value of k that is optimal. This may be due to the higher number of repeats in these genomes which makes it difficult for KmerGenie to find a single best value of k. We can for example see this if we compare Figure 9 and Figure 11 with Figure 13 and Figure 15. These were created in the same way except for the transition matrices that were used. In Figure 13 and Figure 15 (where transition matrix P_2 was used) we can see one or several maxima aside from the one that KmerGenie predicts to be the best. If we compare this to Figure 9 and Figure 11 there is a considerably clearer single maxima in these graphs.

There are a few possible errors that could have caused misleading results in Table 3. One such error is that the genomes were not of exactly identical length and the GC-content and number of repeats could also differ somewhat in the genomes that are compared to each other. Another possible error is that the genome length only is 1000 000 base pairs. A longer genome maybe would have shown different results. This needs further study to determine whether these factors are significant or not.

If we compare the histograms produced by KmerGenie to the histograms of our test they are very different. It is difficult to discover any resemblance, but we can see that when there are clear peaks in the histograms from KmerGenie, we can also see clear peaks in the histograms from our test with a value of k close to the value that KmerGenie predicted. For example in Figure 7 and Figure 8 we can see clear peaks in all of the histograms. This observation indicates that the prediction from KmerGenie is close to the prediction we would have obtained from our histograms, but to be more precise additional histograms have to be made for values closer to the predicted value of KmerGenie.

5.1 Suggested future improvements

Improvements to this experiment can be made by using the command –diploid in KmerGenie. This command should be used when we have two copies of the genome (as we have in this case). This command did not work with the files with reads created in this test because KmerGenie could not find a fit a for any of the histograms.
Another improvement of the test would be to have larger genomes (preferably no less than ten million base pairs) and to simulate them using even more life-like processes to obtain genomes more similar to the real human genome. The genomes could also be simulated in a way that they would have the exact same length and the genomes which were supposed to have the same GC-content actually have the exact same percentage of GC. More runs could be generated to remove stochastic behavior within these simulated experiments and obtain a result that we can be more assured of is correct.
The tests with different k-mer values could be extended to include more values of k, especially close to the value that KmerGenie predicted as the best, to be able to compare them more closely.

With these improvements it would be possible to draw more precise conclusions.

6 Conclusion

We can from the results in Table 3 conclude that all the variables that were tested, repeats, coverage and GC-content, affected KmerGenie's predictions of optimal k as we expected.

From the graphs we can conclude that the result from KmerGenie may not always be the best, since it in many cases shows an uncertainty in the graphs of the result. Especially when reads with a high amount of repeats are to be sequenced, the predicted best value from KmerGenie may be wrong.

From our histograms we can conclude that the prediction of the best value of k from KmerGenie probably is close to the actual best k, but the results are too imprecise to make any further conclusions from these histograms.

The final conclusion from the results of KmerGenie is that coverage and repeats make a significant difference to which value of k that is predicted as being optimal. When a high coverage is used to sequence the reads, a longer length of the k-mers should improve the result (and the opposite for lower coverage). When a genome with many repeats is to be sequenced, a longer k-mer length should be chosen and in the case of a genome with fewer repeats a shorter k-mer length should be used. In the case of many repeats the predicted value calculated by KmerGenie may not be the optimal. If it is important that the result is absolutely as close to reality as possible, the GC-content should also be considered and a longer k-mer length should be chosen if the GC-content is higher than 50% and a lower k-mer length should be chosen if the GC-content is lower than 50%.

References

[1] Scitable. *Nature Education*
Available from: http://www.nature.com/scitable

[2] Illumina (2013). *An Introduction to Next-Generation Sequencing Technology.*
Available from: http://res.illumina.com/documents/products/illumina_sequencing_introduction.pdf

[3] Gios, Ingemar (2008). *Cellbiologisk ordbok*

[4] Pratap K. J. Mohapatra and Rahul Kumar Roy (1991).
Available from: http://www.systemdynamics.org/conferences/1991/proceed/supp-pdfs/moha010.pdf

[5] Sheldon M. Ross (2010). *Introduction to Probability Models* 10th edition

[6] Human Genome Project Information Archive (1990-2003).
Available from: http://web.ornl.gov/sci/techresources/Human_Genome/

[7] Todd, J. Treangen and Steven L. Salzberg (2012). Repetitive DNA and next-generation sequencing: computational challenges and solutions. *Nature* Vol 13, pp 36-46.
Available from: http://www.nature.com/nrg/journal/v13/n1/full/nrg3117.html

[8] Joel L.Cardin (2011). Mutations are the Raw Materials of Evolution. *Nature education* Knowledge 3(10):10
Available from: http://www.nature.com/scitable/knowledge/library/mutations-are-the-raw-materials-of-evolution-17395346

[9] Michael L. Metzker (2010). Sequencing technologies - the next generation. *Nature* Vol 11 pp 31-46.
Available from: http://www.nature.com/nrg/journal/v11/n1/full/nrg2626.html

[10] Joel Baker (2011). De Bruijn graphs and their applications to fault tolerant networks *Oregon State University, scholars archive.*
Available from: http://www.math.oregonstate.edu/~swisherh/JoelBaker.pdf

[11] Phillip E C Compeau, Pavel A Pevzner, Glenn Tesler (2011). How to apply de Bruijn graphs to genome assembly. *Nature* Vol. 29, pp 987-991. Available from: http://www.nature.com/nbt/journal/v29/n11/full/nbt.2023.html

[12] Michael G Ross, et. al. (2013). Characterizing and measuring bias in sequence data. *Genome Biology* doi:10.1186/gb-2013-14-5-r51
Available from: `http://genomebiology.com/2013/14/5/R51`

[13] Chikhi, R. and Medvedev, P. (2013). Informed and Automated k-Mer Size Selection for Genome Assembly *Bioinformatics Advance Access*
Available from: `http://bioinformatics.oxfordjournals.org/content/early/2013/06/02/bioinformatics.btt310.full.pdf`

Appendix

The parameters

Here I present the parameters that was used when running the python programs.

$start_length = 500$
$finish_length = 10000000$

$insert_size = 300$
$coverage = 50 or 100$
$std_dev = 50$
$read_length = 150$
$read_error = 0.05$

$k = 30, 40, 50, 60 or 70$

Python codes

Genome 1

```
import random
import check_GC_content
import argparse
import Genome2_23


# Markov Chain: Transition matrix
transition = {
    'Sub' : {'Sub':0.8, 'TDup':0.08, 'IDup':0.02, 'Ins':0.05,'Inv':0.05},
    'TDup' : {'Sub':0.8, 'TDup':0.06, 'IDup':0.03, 'Ins':0.055,'Inv':0.055},
    'IDup' : {'Sub':0.8, 'TDup':0.085, 'IDup':0.01, 'Ins':0.0525,'Inv':0.0525},
    'Ins' : {'Sub':0.8, 'TDup':0.085, 'IDup':0.025, 'Ins':0.03,'Inv':0.06},
    'Inv' : {'Sub':0.8, 'TDup':0.085, 'IDup':0.025, 'Ins':0.06,'Inv':0.03}}

##transition = {
##    'Sub' : {'Sub':0.1, 'TDup':0.4, 'IDup':0.2, 'Ins':0.1,'Inv':0.2},
##    'TDup' : {'Sub':0.1, 'TDup':0.25, 'IDup':0.25, 'Ins':0.15,'Inv':0.25},
##    'IDup' : {'Sub':0.1, 'TDup':0.45, 'IDup':0.1, 'Ins':0.1,'Inv':0.25},
##    'Ins' : {'Sub':0.1, 'TDup':0.425, 'IDup':0.225, 'Ins':0.05,'Inv':0.2},
##    'Inv' : {'Sub':0.1, 'TDup':0.45, 'IDup':0.25, 'Ins':0.1,'Inv':0.1}}


def Sub(genome):

# Substitutes one nt to another
# Arguments: Genome

    gen = len(genome)
```

```
    number = random.randrange(0,gen)
    genome[number] = generate_bp()
    return(genome)


def TDup(genome):

# Makes tandem duplications; repeats of a sequence next to it
# Arguments: Genome

    gen = len(genome)
    copy = []
    copy_length = abs(int(random.normalvariate(2, gen/20)))
    position = random.randrange(0,gen-copy_length)
    stop = position + copy_length
    copy = [ genome[u] for u in range(position,stop) ]
    genome[stop:stop] = copy
    return(genome)


def IDup(genome):

# Makes an interspersed duplication; repeats of a sequence on another place in
# Arguments: Genome

    gen = len(genome)
    copy = []
    copy_length = abs(int(random.normalvariate(2, gen/20)))
    position = random.randrange(0,gen-copy_length)
    stop = position + copy_length
    copy = [ genome[u] for u in range(position,stop) ]
    inter = random.randrange(0,gen)
    genome[inter:inter] = copy
    return(genome)



def Ins(genome):

# Inserts a sequence
# Arguments: Genome

    gen = len(genome)
    copy_length = abs(int(random.expovariate(1/10.0)))
    copy =  [generate_bp() for i in range(copy_length)]
    inter = random.randrange(0,gen)
    genome[inter:inter] = copy
    return(genome)
```

```
def Inv(genome):

# Inserts a inverted sequence in the genome
# Arguments: Genome

    gen = len(genome)
    copy = []
    copy_length = abs(int(random.normalvariate(2, gen/20)))
    position = random.randrange(0,gen-copy_length)
    stop = position + copy_length
    copy = [genome[nucl] for nucl in range(stop -1, position -1, -1)]
    inter = random.randrange(0,gen)
    genome[inter:inter] = copy
    return genome



def probability_func(state):

# Markov chain - change current state depending on what the last current state
# Argument: dictionary with states and probabilities

    count = 0
    choice = random.random()
    for state_key, prob in state.iteritems():
        count += prob
        if choice <= count:
            current_state = state_key
            return current_state



def new_state(genome, finish_length):

# Sends the genome to the different functions above depending on the current st
# Arguments: The genome

    current_state = 'Sub'
    genome_length = len(genome)
    genome = list(genome)
    counter = 0
    while genome_length < finish_length:
        counter += 1
        if counter % 100 == 0:
            print genome_length
        current_state = probability_func(transition[current_state])
        if current_state == 'Sub':
            genome = Sub(genome)
            genome_length = len(genome)
```

```
        elif current_state == 'TDup':
            genome = TDup(genome)
            genome_length = len(genome)
        elif current_state == 'IDup':
            genome = IDup(genome)
            genome_length = len(genome)
        elif current_state == 'Ins':
            genome = Ins(genome)
            genome_length = len(genome)
        elif current_state == 'Inv':
            genome = Inv(genome)
            genome_length = len(genome)
        else:
            print('something is wrong')
    return genome



bps = {'A' : 0.35,'C' : 0.15, 'G' : 0.15, 'T' : 0.35}

def generate_bp():
    count = 0
    choice = random.random()
    for bp, prob in bps.iteritems():
        count += prob
        if choice <= count:
            return bp


def main(args):

# Makes a random genome with 500 nt and then increases the length of the genome
# Arguments: length of the random genome, length of the finished genome)

    genome = ''.join([generate_bp() for i in range(args.start_length)])

    nucleobases = ['A','C','G','T']
##    i = 0
##    genome = ''
##    while i < args.start_length:
##        rand = random.choice(nucleobases)
##        genome = genome + rand
##        i += 1
    length_genome = len(genome)
    print ('length of genome:', length_genome)
    genome = new_state(genome, args.finish_length)


    genome_str = "".join(genome)
    outfile = open(args.outfile+'1.fa','w')
```

```
    print >>outfile, genome_str

    GC_content = check_GC_content.check_GC_content(genome)
    print GC_content

    genome2_str = Genome2_23.diploid(genome, 0.0001, 0.0001, 0.0001)
    outfile2 = open(args.outfile+'2.fa','w')
    print >>outfile2, genome2_str



if __name__ == '__main__':

# Take care of input

    parser = argparse.ArgumentParser(description = "Simulate a genome of desire
    parser.add_argument('start_length', type=int, help='The length of the first
    parser.add_argument('finish_length', type=int, help='The approximate size o
    parser.add_argument('outfile', type=str, help='outfile prefix. ')

    args = parser.parse_args()
    main(args)
```

Genome 2

```
import random

def insertion():
     return(''.join([random.choice('AGCT') for i in range(random.randint(1,10))

def deletion():
    return(abs(int(random.gauss(4,2))))

def mutation():
    return(random.choice('AGCT'))

def diploid(genome,insertion_rate, deletion_rate, mutation_rate):
    genome_copy = []
    i = 0
    while i < len(genome):
        if random.uniform(0,1) < mutation_rate:
            genome_copy.append(mutation())
            #print 'here'
        elif random.uniform(0,1) < deletion_rate:
            i+= deletion()
            #print 'here2'
        elif random.uniform(0,1) < insertion_rate:
            genome_copy.append(insertion())
            #print 'here3'
```

```
        else:
            genome_copy.append(genome[i])
        i+=1
    return ''.join([nucl for nucl in genome_copy])
```

Reads

```
import os
import random
import argparse
import check_GC_content
import numpy


def reverse_complement(string):

# Reverse complements of a DNA-string
# Arguments: A DNA string, Returns: A python string that represents the

    rev_nuc={'A':'T','C':'G','G':'C','T':'A','N':'N','X':'X'}
    rev_comp = ''.join([rev_nuc[nucl] for nucl in reversed(string)])
    return(rev_comp)



def GC_content(fragment1, fragment2):

    GC_percentage1 = check_GC_content.check_GC_content(fragment1)
    GC_percentage2 = check_GC_content.check_GC_content(fragment2)

    if GC_percentage1 <= 20:
        probability1 = 1
    elif GC_percentage1 >= 80:
        probability1 = 0.58
    else:
        probability1 = (-0.7*GC_percentage1+114)/100


    if GC_percentage2 <= 20:
        probability2 = 1
    elif GC_percentage2 >= 80:
        probability2 = 0.58
    else:
        probability2 = (-0.7*GC_percentage2+114)/100

    read_fragment1 = numpy.random.random() < probability1
    read_fragment2 = numpy.random.random() < probability2

    return (read_fragment1, read_fragment2)
```

```
def print_read(pe_1,pe_2,genome,fragment_start_pos,fragment_length, read_length

# Prints the paired end reads
# Arguments: file with paired ends 1, file with paired ends 2, start position,

    fragment1 = genome[fragment_start_pos:fragment_start_pos+read_length]
    fragment2 = reverse_complement(genome[fragment_start_pos+fragment_length-re

    if len(fragment1) == read_length and len(fragment2) == read_length:
        (read_fragment1, read_fragment2) = GC_content(fragment1, fragment2)

        if read_fragment1 == True and read_fragment2 == True:

            fragment_1 = list(fragment1)
            fragment_2 = list(fragment2)

            fragment_list = fragment_1+fragment_2
            read_len = len(fragment_list)

            basepairs = ['A','C','G','T']
            mu = 1/read_error
            start_place = random.randint(0,mu)
            fragment_list[start_place] = random.choice(basepairs)
            error_place = int(random.gauss(mu,40))

            while error_place < read_len:
                fragment_list[error_place] = random.choice(basepairs)
                new_error_place = int(random.expovariate(1/mu))
                error_place += new_error_place

            fragments = "".join(fragment_list)
            fragment1, fragment2 = fragments[:len(fragments)/2], fragments[len(

            qual_fragment1 = 'J'*read_length #random.choice['ABCDE']
            qual_fragment2 = 'J'*read_length

            print >>pe_1,('@read_1'+str(read_number)+'genome_copy:'+str(genome_
            + 'read_1'+str(read_number)+'genome_copy:'+str(genome_copy)+'\n'+ q

            print >>pe_2,('@read_2'+str(read_number)+'genome_copy:'+str(genome_
            + 'read_2'+str(read_number)+'genome_copy:'+str(genome_copy)+'\n'+ q

            return read_number + 1
        else:
            return read_number

    else:
```

```
        return read_number


def main(args):

# Makes paired end reads and save them in two seperate files.
# Arguments: insertsize, coverage, standard deviation, length of reads, the gen

    genome1 = open(args.file_genome,'r')
    genome11 = open(args.file_genome2,'r')
    genome = genome1.read().strip()
    genome2 = genome11.read().strip()
    genome_length = len(genome)

    number_of_reads=(genome_length*args.coverage)/(2*args.read_length)
#Specifiels the number of simulated read pairs (related to insertion size lengt
    print ('number_of_reads', number_of_reads)

    if not os.path.exists(args.outfolder):
        os.makedirs(args.outfolder)

    pe_1=open(os.path.join(args.outfolder,'PE_1.fa') ,'w')
    pe_2=open(os.path.join(args.outfolder,'PE_2.fa'),'w')

    passed_reads = 0

    while passed_reads <= number_of_reads:
        location_on_genome = random.randint(0,genome_length)
        fragment_length = int(random.gauss(args.insertsize,args.std_dev))

        if (location_on_genome + fragment_length >= genome_length):
            continue
        else:
            parameter = random.choice([0,1])

            if parameter == 0:
                passed_reads = print_read(pe_1, pe_2,genome,location_on_genome
            elif parameter == 1:
                passed_reads = print_read(pe_1, pe_2,genome2,location_on_genome
            else:
                print('something is wrong')

        if passed_reads % 10000 ==0:
            print passed_reads



if __name__ == '__main__':
```

```
# sim_out = open(os.path.join(args.outpath,'bwa_vs_getdristr_sim_out'),'w')

    parser = argparse.ArgumentParser(description = "Simulate k-mers from reads,
    parser.add_argument('file_genome', type=str, help='file with the genome')
    parser.add_argument('file_genome2', type=str, help='file with the other gen
    parser.add_argument('insertsize', type=int, help='...')
    parser.add_argument('coverage', type=int, help='The coverage')
    parser.add_argument('std_dev', type=int, help='The standard deviation')
    parser.add_argument('read_length', type=int, help='the length of the reads'
    parser.add_argument('read_error', type=float, help='how many percentage of
    parser.add_argument('outfolder', type=str, help='location of the folter to

    args = parser.parse_args()

    main(args)
```

Kmers

```
import matplotlib.pyplot as plt
import numpy
import sys
import argparse


def make_kmers(k, read, read_comp, kmer_dict):

# Take one read and split it into kmers of the size that you decide, and then c
# Arguments: size of kmer, the read, a dictionary with the kmers

    length = len(read)
    while length >= k:
        kmer1 = read[:k]
        kmer2 = read_comp[:k]
        if kmer1 in kmer_dict:
            kmer_dict[kmer1]+=1
        else:
            kmer_dict[kmer1]=1
        if kmer2 in kmer_dict:
            kmer_dict[kmer2]+=1
        else:
            kmer_dict[kmer2]=1
        read = read[1:]
        read_comp = read_comp[1:]
        length = len(read)



def kmer_plot(kmers):
```

```
# Makes a plot from the list with numbers of kmers
# Arguments: the list with number of kmers

    plt.hist(kmers, 1000)
    plt.axis([0, 800, 0, 350000])
    plt.grid(True)
##    plt.show
    plt.savefig('../data/genome1000000/genome1/transition_1/coverage50/k30/kme



def reverse_complement(string):

# Reverse complements of a DNA-string
# Arguments: A DNA string, Returns: A python string that represents the

    rev_nuc={'A':'T','C':'G','G':'C','T':'A','N':'N','X':'X'}
    rev_comp = ''.join([rev_nuc[nucl] for nucl in reversed(string)])
    return(rev_comp)



def main(args):

# Main program that takes out the reads from the fastaq-format and sends them t
# Arguments: size of kmers

    reads1 = open(args.file_reads1,'r')
    reads2 = open(args.file_reads2,'r')

    k = args.k
    read_row = 0
    kmer_dict = {}

    for line in reads1:
        if read_row == 1:
            read1 = line.strip()
            read1_comp = reverse_complement(read1)
            make_kmers(k, read1, read1_comp, kmer_dict)
            read_row = 0
        if line[0] == "@":
            read_row = 1

    for line in reads2:
        if read_row == 1:
            read2 = line.strip()
            read2_comp = reverse_complement(read2)
            make_kmers(k, read2, read2_comp, kmer_dict)
            read_row = 0
        if line[0] == "@":
```

```
            read_row = 1

    count_kmers = kmer_dict.values()

    filename = '../data/genome1000000/genome1/transition_1/coverage50/k30/kmers
    outfile = open(filename,'a')

    for j in count_kmers:
        print >>outfile, j

    kmer_plot(count_kmers)



if __name__ == '__main__':

    parser = argparse.ArgumentParser(description = "Simulate k-mers from reads,
    parser.add_argument('file_reads1', type=str, help='file with the reads in f
    parser.add_argument('file_reads2', type=str, help='file with the reads in f
    parser.add_argument('k', type=int, help='size of the k-mers')

    args = parser.parse_args()
    main(args)
```

GC-content

```
def check_GC_content(genome):
    length = len(genome)
    #print length

    countG = genome.count('G')
    countC = genome.count('C')
    countT = genome.count('T')
    countA = genome.count('A')


    percentG = 100*(countG/float(length))
    percentC = 100*(countC/float(length))
    percentT = 100*(countT/float(length))
    percentA = 100*(countA/float(length))


    if round(percentG+percentC+percentT+percentA) == 100:
        percentage = percentG + percentC
        return percentage
    else:
        print ('something is wrong')
        print ('total percentage', percentG+percentC+percentT+percentA)
```

Plots from KmerGenie and our experiments

Figure 1: KmerGenie Genome1, coverage=50

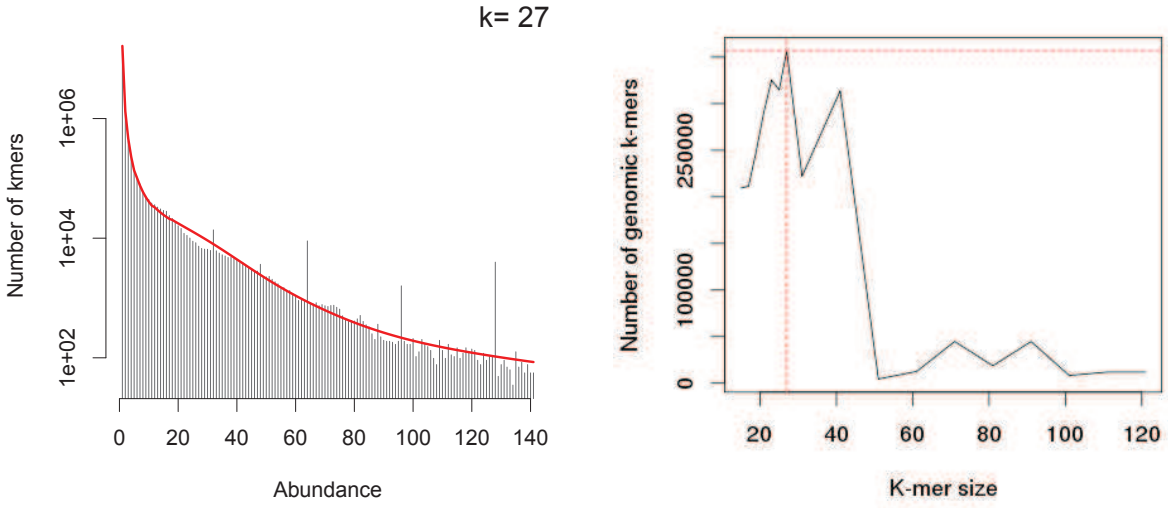
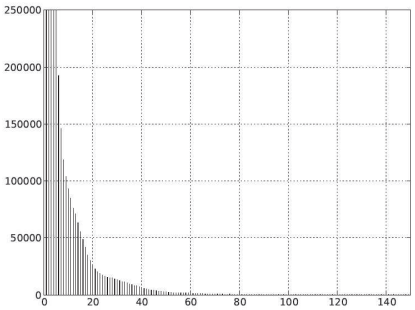
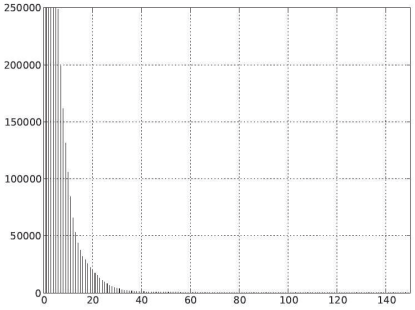


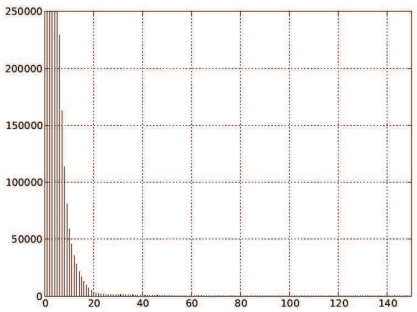
Figure 2: Our test Genome1, coverage50



(a) k = 30



(b) k = 40



(c) k = 50

Figure 3: KmerGenie Genome1 coverage = 100

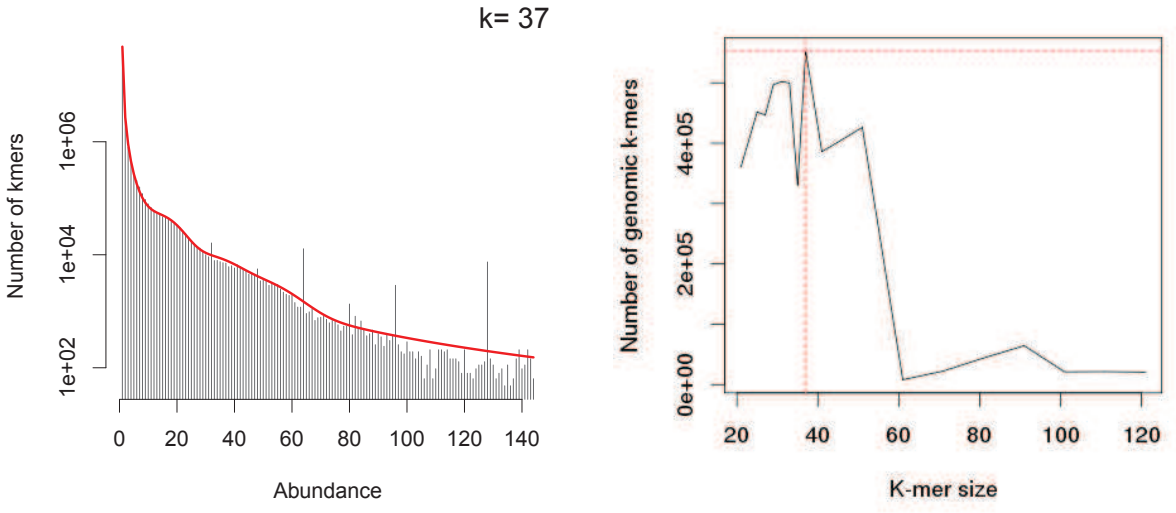
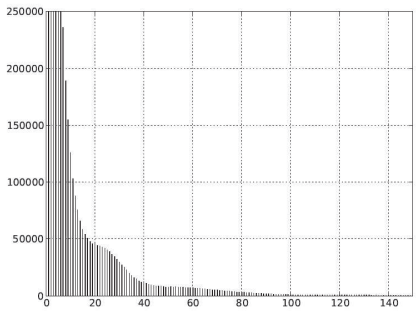
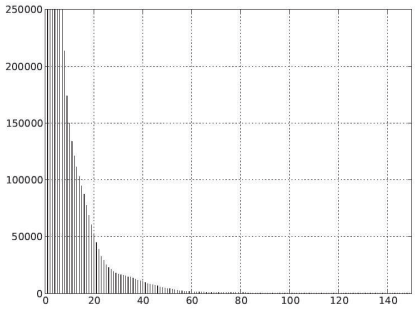


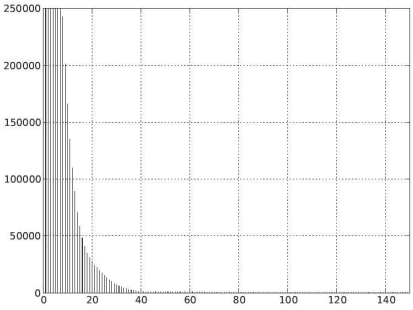
Figure 4: Our test Genome1 coverage = 100



(a) k = 30



(b) k = 40



(c) k = 50

Figure 5: KmerGenie Genome2 coverage = 50

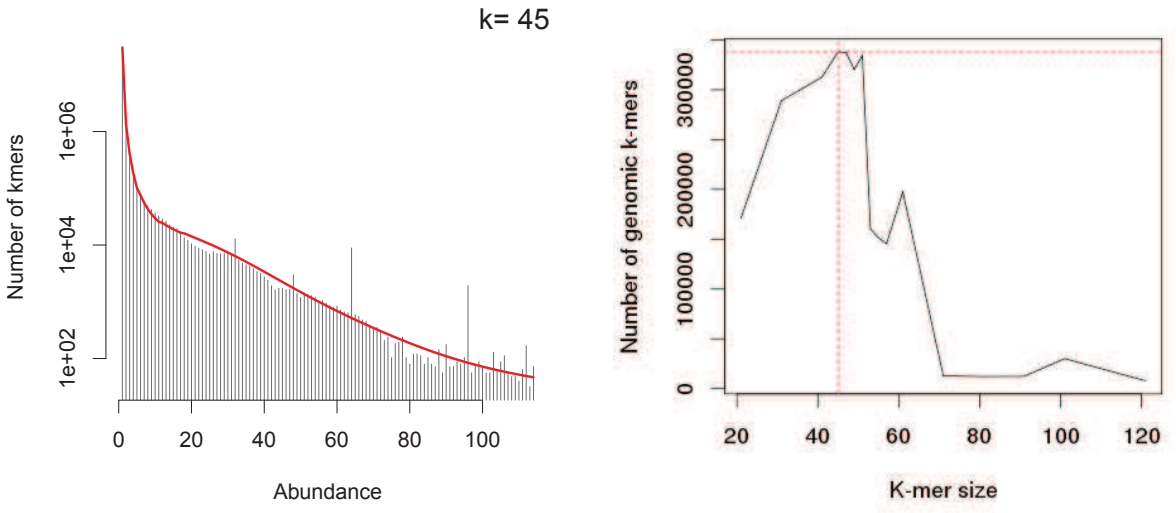
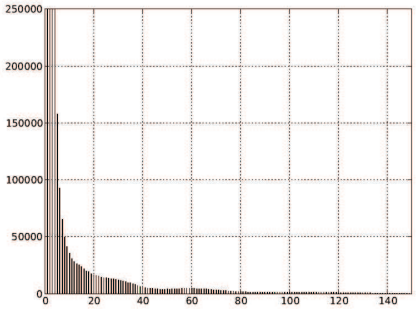
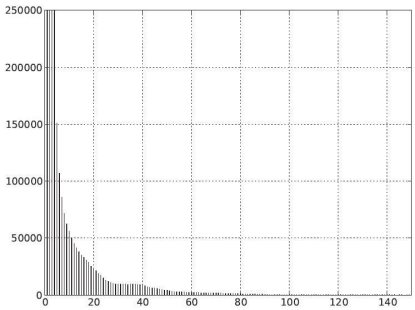


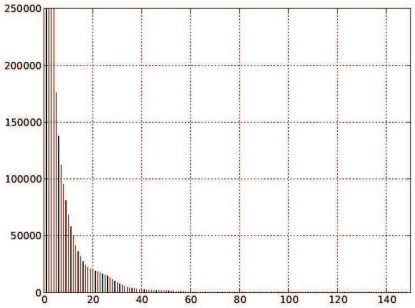
Figure 6: Our test Genome2 coverage = 50



(a) k = 30



(b) k = 40



(c) k = 50

Figure 7: KmerGenie Genome2 coverage = 100

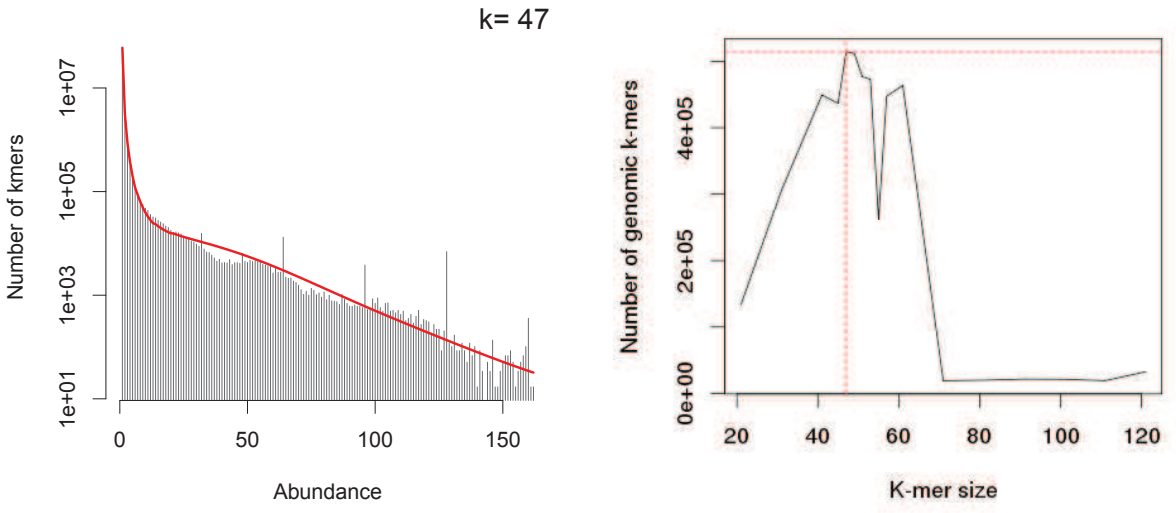
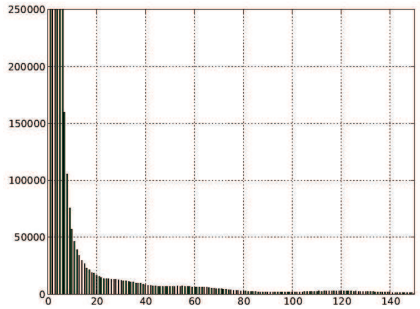
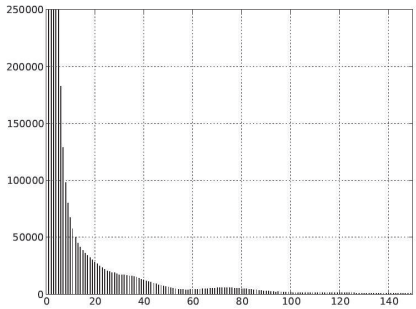


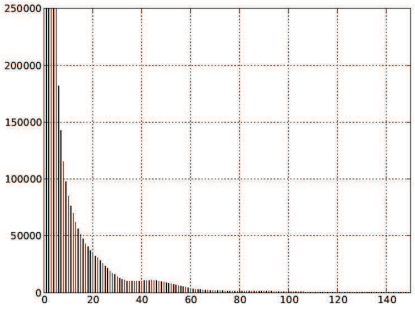
Figure 8: Our test Genome2 coverage = 100



(a) k = 30



(b) k = 40



(c) k = 50

Figure 9: KmerGenie Genome3 coverage = 50

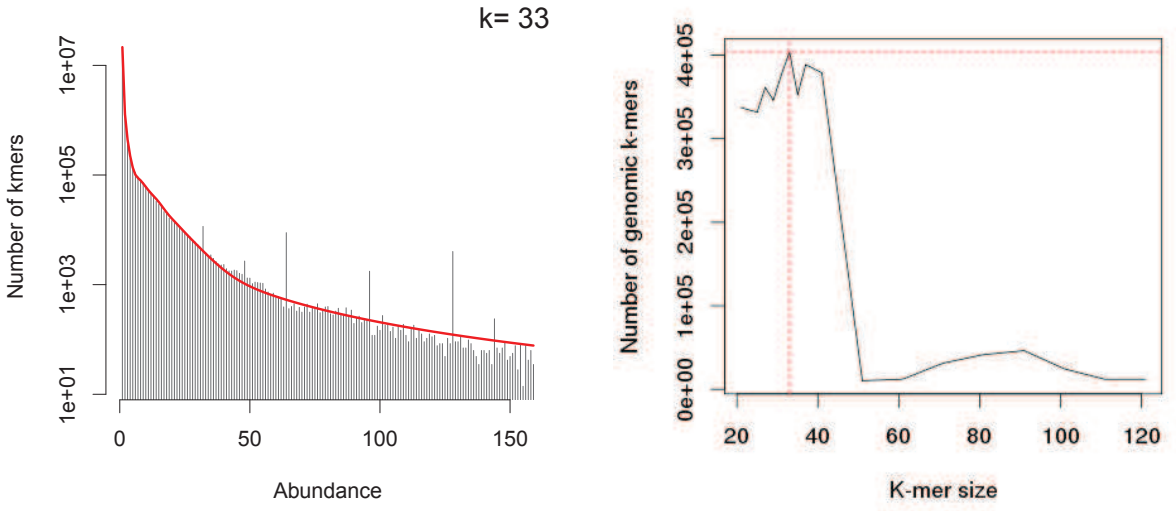
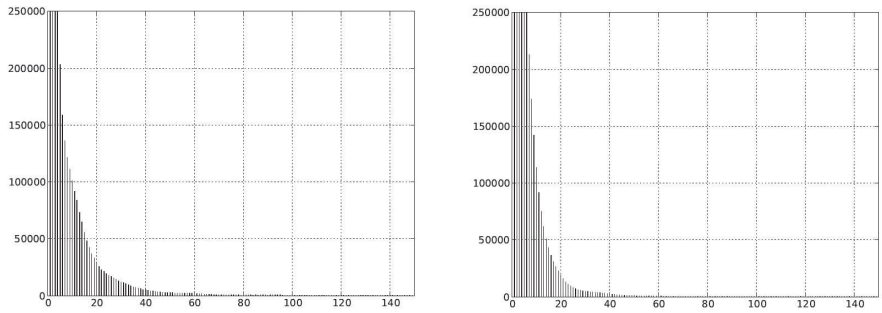
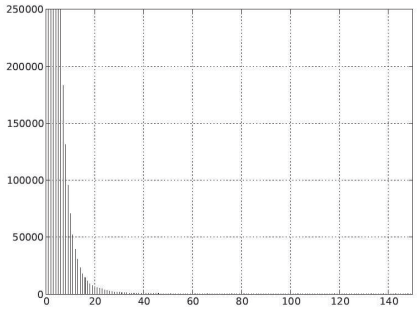


Figure 10: Our test Genome3 coverage = 50



(a) k = 30

(b) k = 40



(c) k = 50

Figure 11: KmerGenie Genome3 coverage = 100

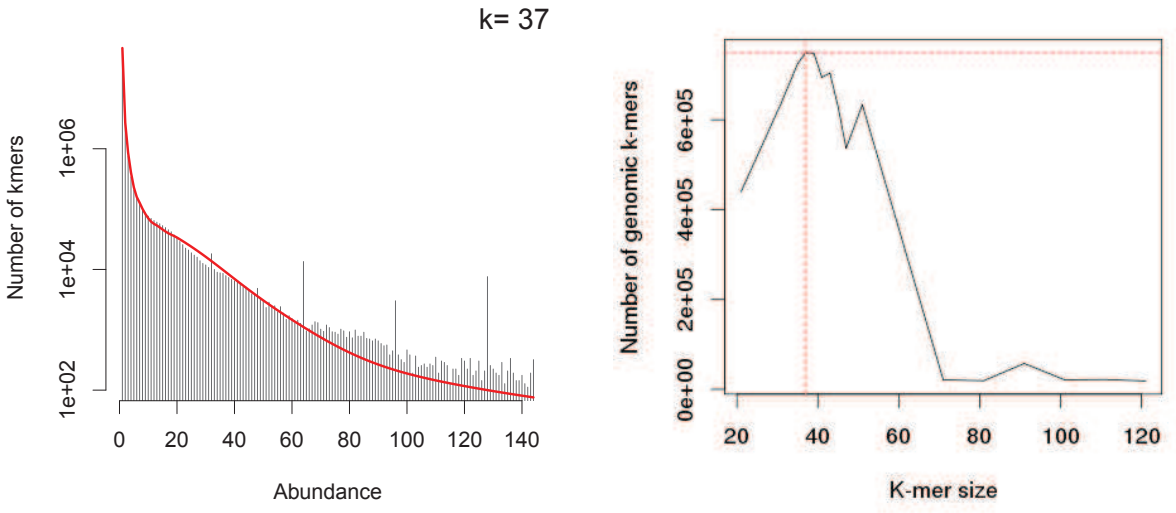
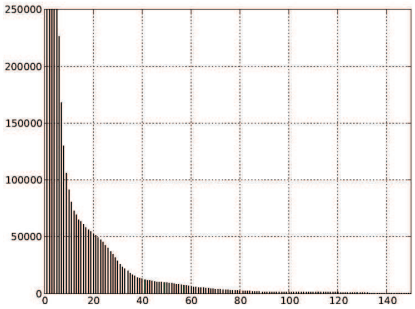
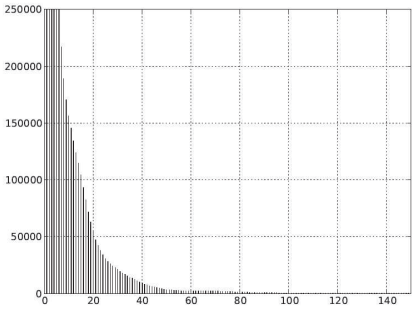


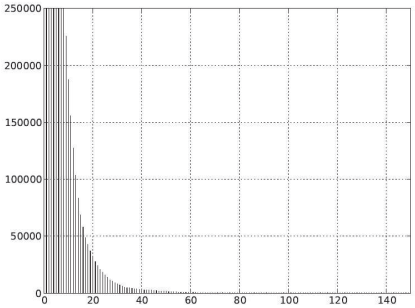
Figure 12: Our test Genome3 coverage = 100



(a) k = 30



(b) k = 40



(c) k = 50

Figure 13: KmerGenie Genome4 coverage = 50

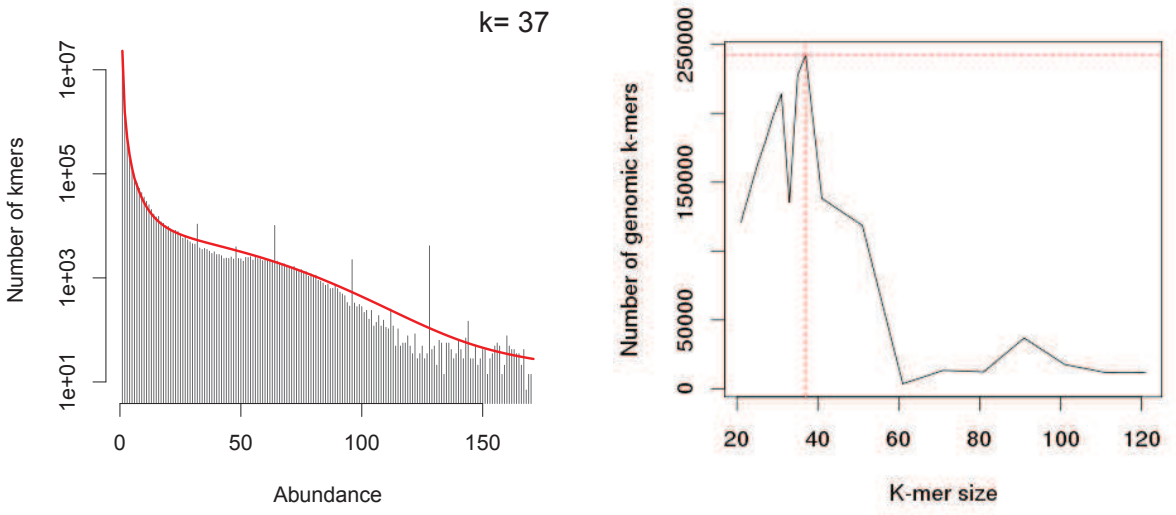
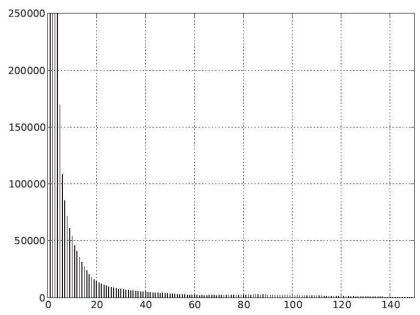
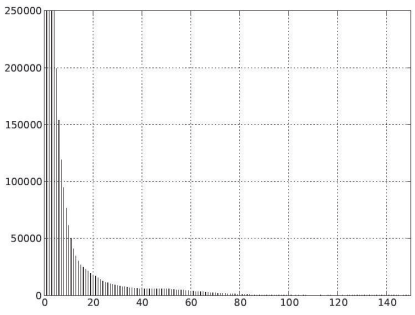


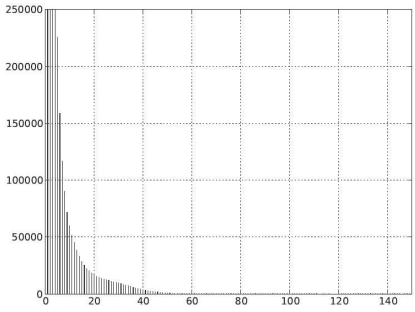
Figure 14: Our test Genome4 coverage = 50



(a) k = 30



(b) k = 40



(c) k = 50

Figure 15: KmerGenie Genome4 coverage = 100

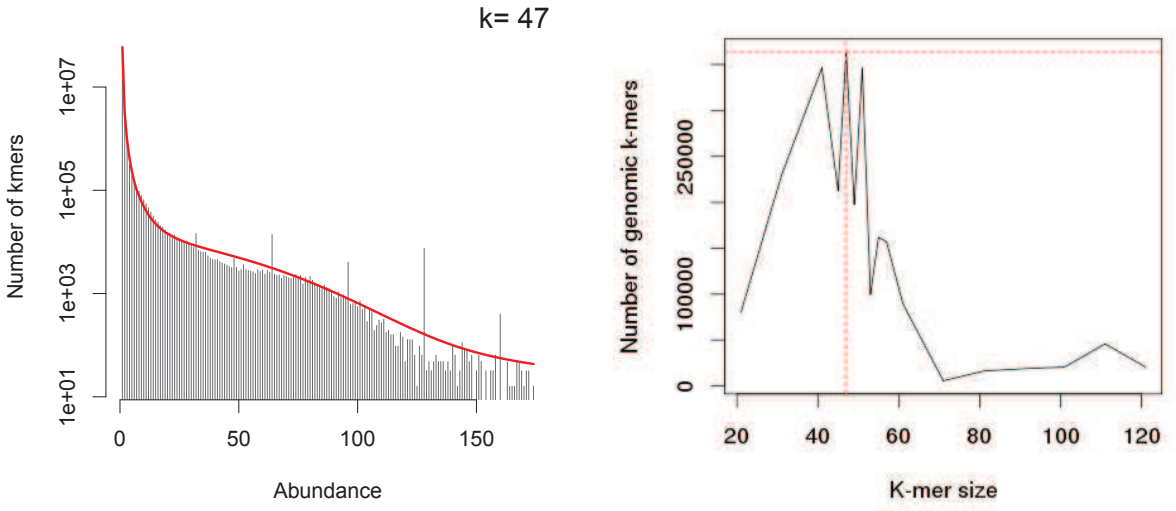
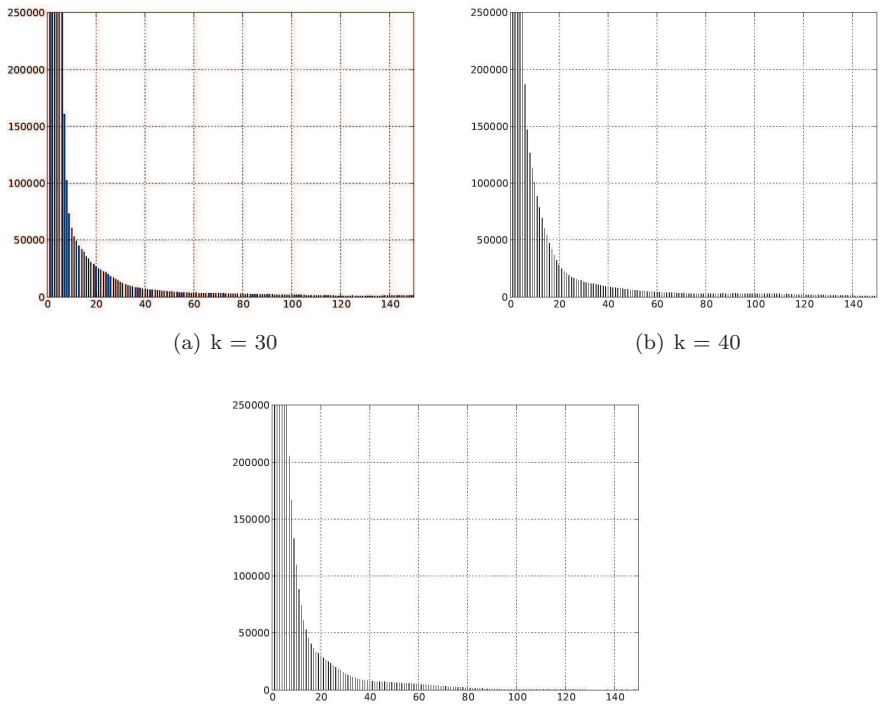


Figure 16: Our test Genome4 coverage = 100



(a) k = 30

(b) k = 40

(c) k = 50

Figure 17: KmerGenie Genome5 coverage = 50

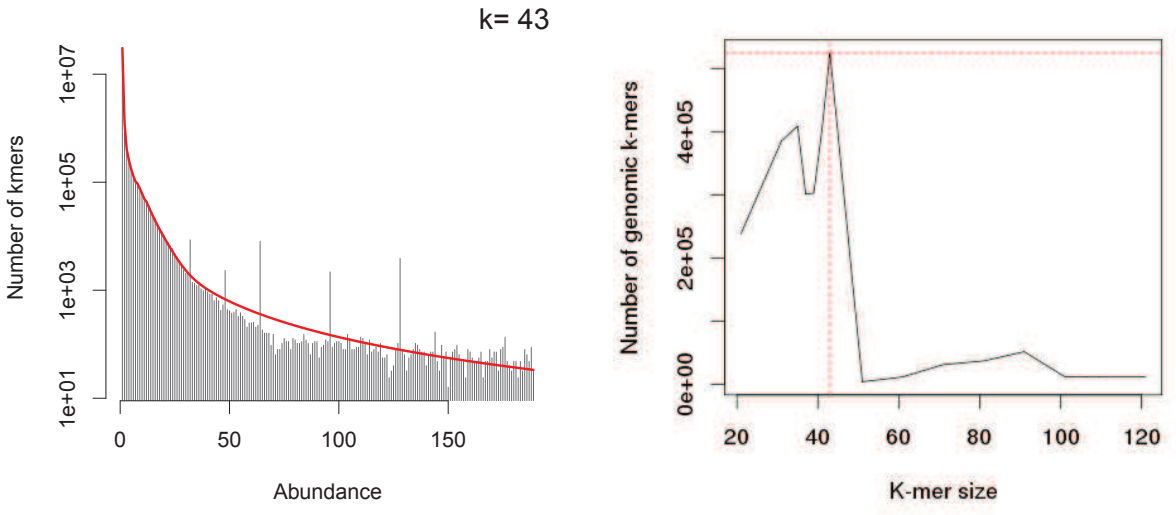
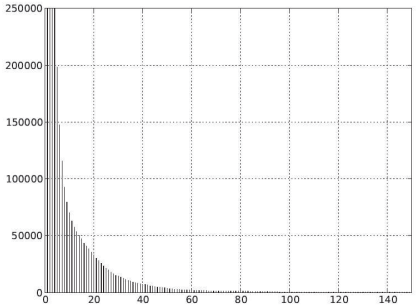
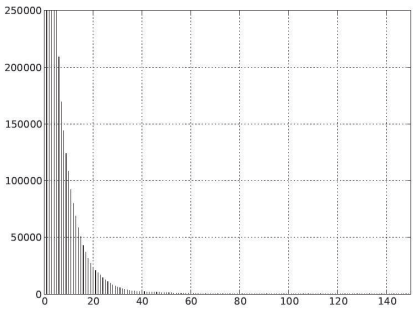


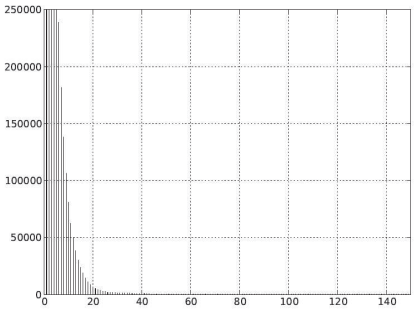
Figure 18: Our test Genome5 coverage = 50



(a) k = 30



(b) k = 40



(c) k = 50

Figure 19: KmerGenie Genome5 coverage = 100

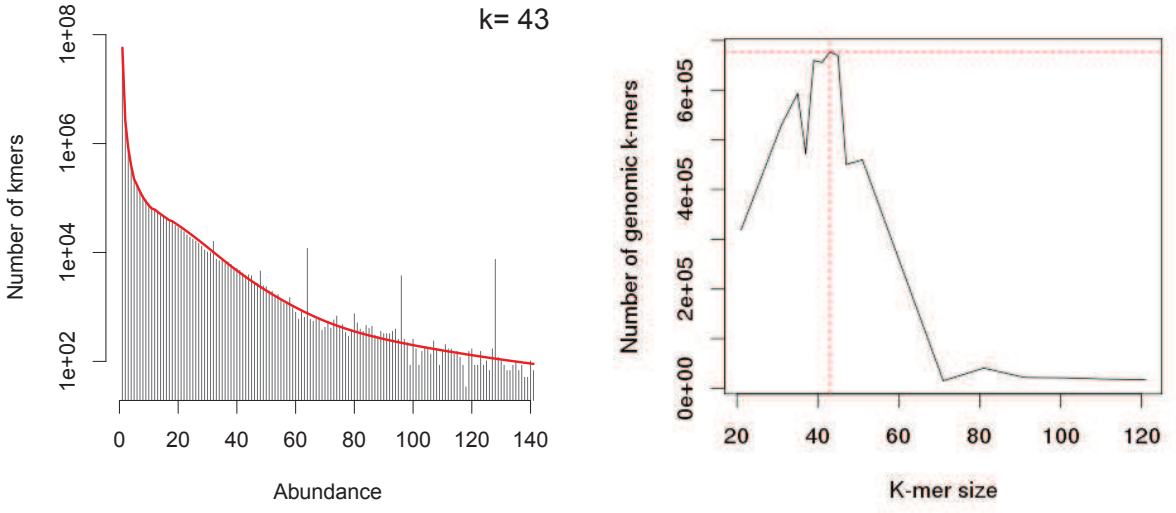
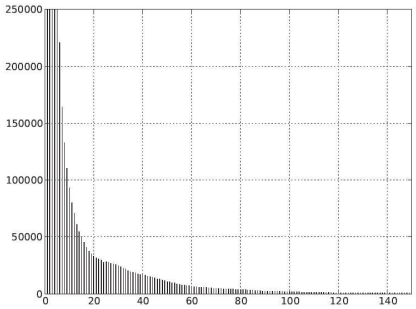
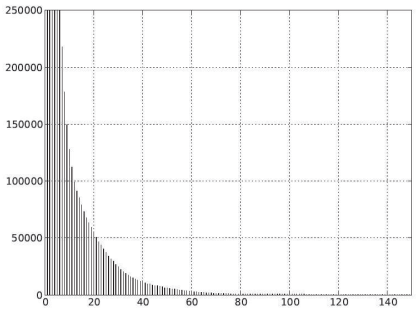


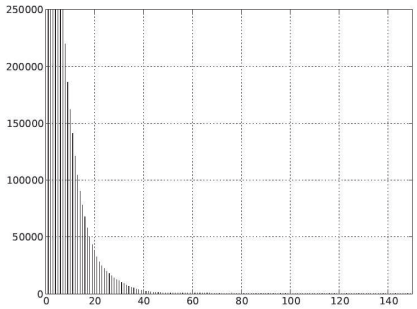
Figure 20: Our test Genome5 coverage = 100



(a) k = 30



(b) k = 40



(c) k = 50

Figure 21: KmerGenie Genome6 coverage = 50

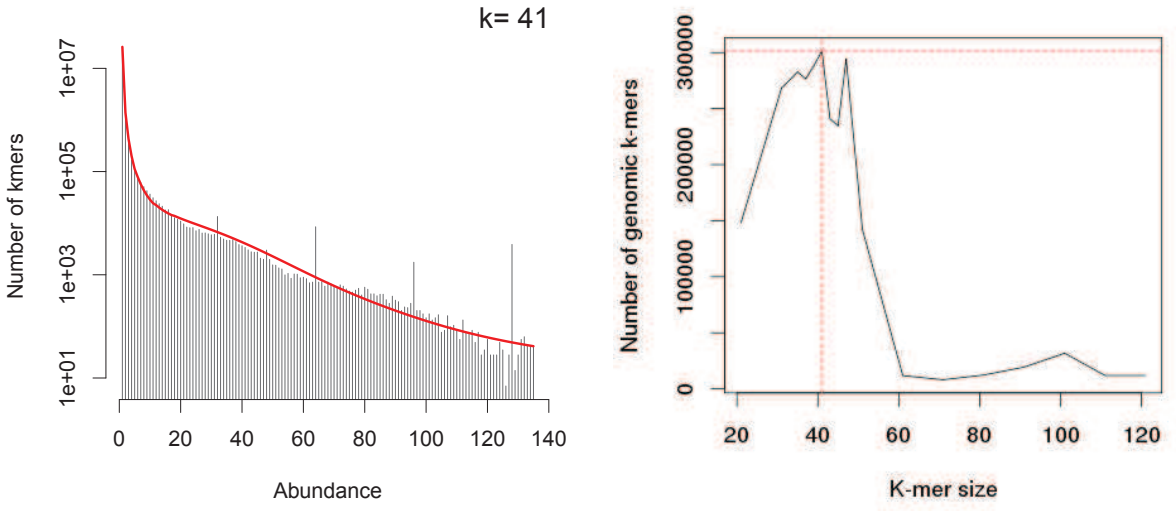
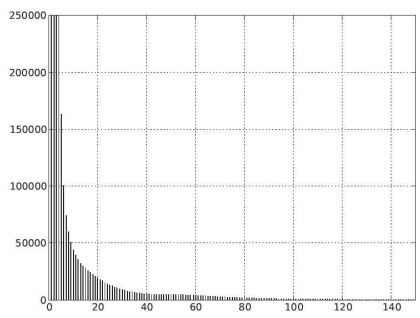
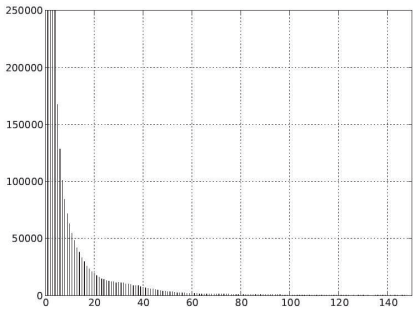


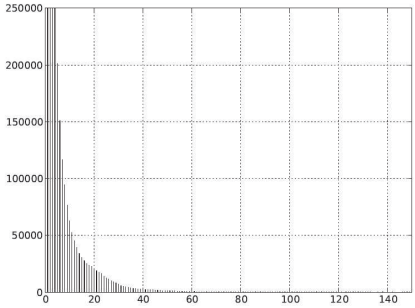
Figure 22: Our test Genome6 coverage = 50



(a) k = 30



(b) k = 40



(c) k = 50

Figure 23: KmerGenie Genome6 coverage = 100

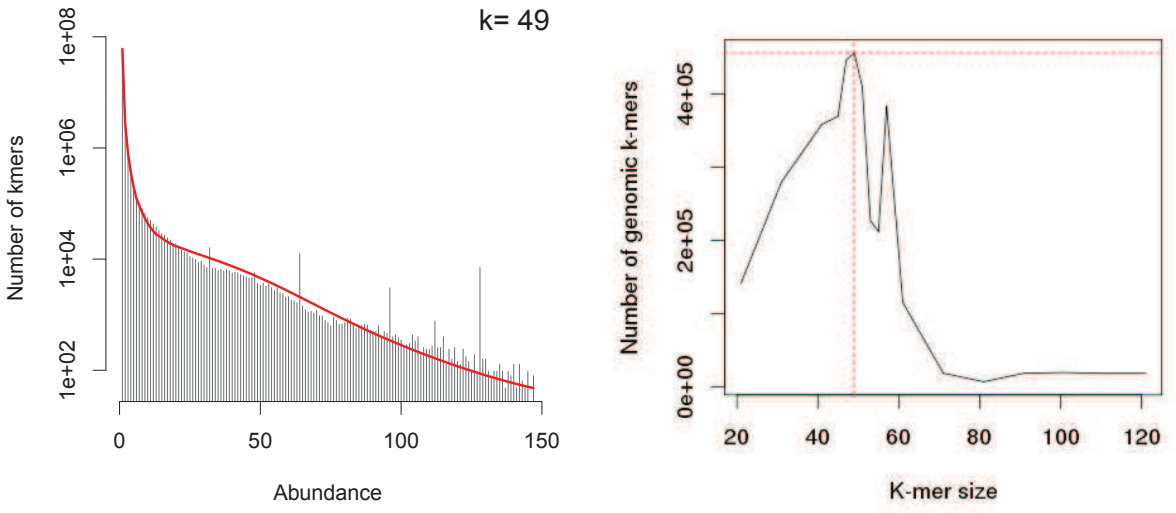
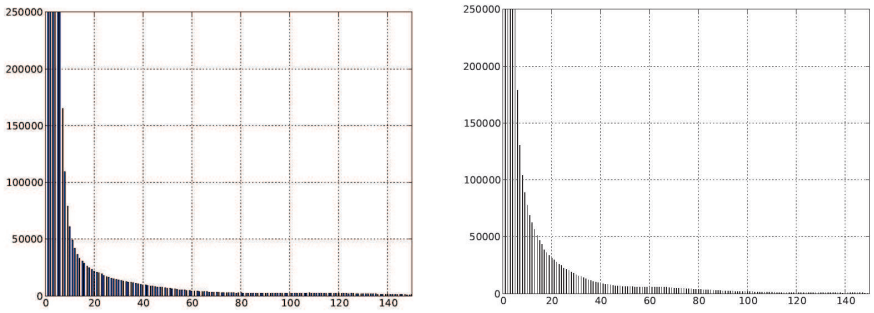
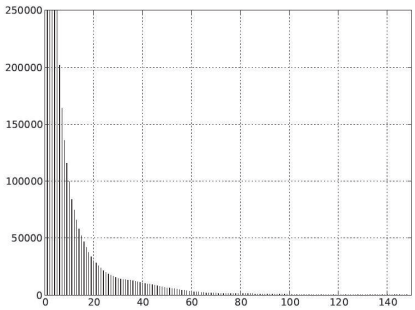


Figure 24: Our test Genome6 coverage = 100



(a) k = 30

(b) k = 40



(c) k = 50