Report

Introduction

The goal of the project is to assemble the sequenced reads of an organism genome by using a de Bruijn graph. We start with raw reads and are supposed to give an assembled genome or at least the minimal number of contigs represents him. The process begins by parsing the reads, extracting all kmers of a given size, then creating a de bruijn graph to get contigs from it.

How to optimize memory and performances

Memory

The first obvious point is that we analyze compressed reads without unpacking them. When we parse the compressed reads, we store none, only the kmer that we'll use. The kmer size is relatively long, it allows us to store a lot more less.

Performances optimisation

Kmer size management

The kmer long size also avoids testing irrelevant branches in the graph because of genomic repetition and computing a lot more than needed.

Streaming treatment

To bring the script to new age of rapidity, we don't create the whole de bruijn graph, as we extend the contig, we delete tips and manage bubbles in streaming so we don't have to clean a whole polluted de bruijn graph

Data structures to store informations

To store the kmers we had to choose between hash table and bloom filter. The goal is to test the presence of an overlapping kmer by k-1 to a given kmer of the same size k.

We decided to use a python dictionary as a hash table because it has a more practical usage such as keeping the number of apparitions of the kmer and adding '-' before that number when we use the kmer in the assembly. It allows us not to lose track of the number and notify us that we won't use it anymore.

Abandoned concepts

Deleting kmers count

We started by deleting the kmer used in the dictionary but we had to use it to choose among the branches of the bubble. To not be arbitrary in the choice, we weigh each branch of the bubble by summing the number of apparitions of the kmers composing the branch, if a branch is way "heavier" it will be kept because its probability to be the right one is bigger.

Recursive way to form the contigs

The first way we tried to extend the contig, was to recursively search for the successors. In short, when trying to find the successor of the first kmer, it searched for the successor of the successor etc. It didn't work

Choice in the bubble

With quast, we could test the best parameters in the choice process of the most probable branch of the bubble. At first we wanted to make the choice at all cost and took the "deepest" branch, the quast results was bad because it caused misassemblies.

After some changes we saw that below a certain value of significant difference, we couldn't choose between the two branches when the smallest depth was above 1/5th of the biggest depth depth.

Arbitrary values choices

Filter

All kmer that are present less than 3 times are considered errors so we filter them.

Tips

We consider a branch that is smaller than 1/3 than the other in the fork as a tip and then is deleted.

Kmer size

The Kmer size by default is 80 because of the maximum size of the repetition in the lambda phage. Getting kmers bigger than any repetition allows us to not connect wrongly kmers together.

Results and interpretation

Results

	Genome frac	Duplication ratio	Contig nb	Misassembly nb	Misass. length
Phage without error	99.899 %	1	1	0	0
with error	99.769 %	1.002	1	0	0
Medium	99.94 %	1.035	9	2	6466
Hard	64.36 %	1.06	14	2	5113

Interpretation

The phage genome is way easier to assemble than the medium and hard ones, we got no misassembly for the two phage genomes. For the medium, we got a very high coverage rate but the sum of length of the misassembled contigs is higher than in the hard. To sum it up, our assembler is enough to cover a genome that is adapted to assembly, but is limited against trials like the Hard genome.