

INFO-F439 - ADVANCED METHODS IN BIOINFORMATICS :
PROJECT REPORT

Read Mapping

Professors:


Matthieu DEFRANCE

Wim VRANKEN

Sassine NAIM

Matricule 440091

May 2021



Contents

1	Introduction	2
2	Algorithm	3
2.1	Algorithm Description	3
2.2	Code Description	4
2.2.1	Hashing the Genome	5
2.2.2	Implementing Seed and Extend	5
3	Variable Selection	6
4	Results and Conclusions	7

1 Introduction

Technologies being developed in the field of Bioinformatics and genomics allow us today, to determine the sequence of nucleotides of many and millions of DNA molecules in parallel. These technologies are allowing us today to not only take a closer look at DNA molecules and their composition, but also to conduct studies in different fields, like studying human genetic diseases, or analysing the interaction between proteins and the DNA, or even conduct unsupervised machine learning studies on the data to generate new insights and results. But reading DNA sequences usually comes with a catch : we can't read the entire genome at once, so the data generated by these technologies usually comprises huge numbers of very short DNA sequences, that we call 'reads'. These reads need therefore, to be pieced together in a certain way to form the entire genome. A way to solve this complex puzzle, is Read Mapping. This technique consists in mapping the reads back to a reference genome to determine their positional origin.

Many different algorithms exist to accomplish read mapping. In this project, it was asked to implement a read mapping software that is able to align a large set of reads to a given genome. The reads that will be used to test the algorithm are generated using ChIP-seq, and the genome is the one of a *Drosophila melanogaster* "Oregon-R S2" strain. During the testing of our algorithm, I used a reduced dataset, since the original one is of a non negligible size and takes a long time to compute.

In this particular report, I will start by introducing the algorithm I decided to implement. I will then go into details about the different parts of the implementations, more specifically variable selection. Finally, I will end this report by some results and conclusions.

2 Algorithm

2.1 Algorithm Description

The algorithm here used is the Seed and Extend algorithm. This particular algorithm was introduced in class, but after doing some research online, I realised that many forms of this algorithm were available. Here is the version that I decided to implement :

- First, we index the genome. Here, many different forms of data storing types were available. I decided to go with a hash table. When hashing the genome, we use k-mers of same size to cut through the genome and insert it into a hash table.
- Second, we create seeds. To create seeds, we take each and every read on its own. We also split the read, just like the genome, into k-mers. The size of the seeds as well as the interval at which they are taken are variables to be selected, I will talk more about that later on.
- After taking a specific read, and splitting it into different seeds, we loop over all the seeds of a specific read, and for every seed we send a query to the previously created hash table with the seed as argument. This query will return a list of the places where this seed is seen in the genome. This list would be empty of course if nothing matched the seed in the genome.
- Now that we have, for each read, a list of seeds, and for each seed its matching k-mers within the genome (with their position within the genome), we have finished the "Seed" part of the "Seed and Extend" algorithm.
- For the extend part, and here is where I realised a lot of methods were available, here is what I did.
- What we want here is select the best seeds of every read, so the seeds that represent the most accurately the read within the genome. That is exactly why its called read mapping. We take the reads, and map them to the genome.
- For each read, we take its different seeds. For each seed of a read, we have three possibilities. Either this seed was matched to multiple k-mers of the genome, either it was matched to one k-mer, and either to no k-mers at all.

- To select the best seeds of this particular read, I implemented a technique that is very close to the one we have seen in class, called "match and extend".
- What I do is, for each seed that we take, if this seed doesn't have matching k-mers with the genome, the job is done here, we can already decide not to choose this seed. But if it does (which is usually the case) :
 - I align this seed with the corresponding k-mer in the genome
 - I lookup left, and look up right from both, the seed and its matching k-mer in the genome
 - If the newly looked upon nucleotides are the same, then I don't penalise
 - If any of the newly looked upon nucleotides is not the same, whether its left or right or both, I subtract -1 (or -2 if both left and right) to a score that is later assigned to this matching between the seed and the k-mer
 - I continue on doing that, looking left and right till I extend my seed to my read
- What is explained above gives me access to a score particularising each seed of a read. This score represents the "match-ability" between the read and this particular place in the genome that was detected by matching a seed to a k-mer.
- Since I subtract 1 every time I have a mismatch, the higher the score, the closer is the read to this place in the genome.
- As you can imagine now, each seed will have its own score.
- Remember, the goal here is to choose the seeds that will represent in the best way possible the read they belong to. To do that, I sorted the different seeds for each read, according to the obtained score. Then I took the 5% of the seeds with the highest score and trimmed to have only seeds with score that is bigger than -3 . The choice of this variable will be discussed later on.
- As an output, we'll have a dictionary, that contains for every read, the list of seeds that matched to a particular place in the genome the best, by representing their read.

2.2 Code Description

To get into more details, and to provide a better visualisation and explanation of the algorithm that I chose to implement, I will describe in this section how I coded this algorithm.

I also commented my code so that it would be easier to understand it.

2.2.1 Hashing the Genome

The first class I created is to index the genome into a hash table. To do that, I need to create an object of the *Genome_Hashing* class and give it as argument the sequence as a string, the size of the k-mers and the interval at which I want to sequence the genome. When I initialise the object *Genome_Hashing*, the hash table is created. This table is created by going through the entire genome, and storing the k-mer in the hash table. Of course, similar k-mers will have the same key in the table (which is the point of an indexing like that) but will also contain the different positions at which this k-mer is available in the genome.

Here is what the hashing table could look like (let's say size of k-mers is 4):

- "AGCT" -> 4, 12
- "GCTA" -> 18, 25
- "CCTA" -> 40, 50

I also implemented a *query* method that given the k-mer, it will return the positions at which it is present in the genome.

2.2.2 Implementing Seed and Extend

- For the seed part of the algorithm, we first parse the genome file and the reads file. By doing that we store the genome in a string and the reads in a dictionary with information and variables assigned to each read.
- We then call the *seed()* function. This function will index the genome using the previously introduced class and will create the seeds based on this indexing. The seeds are created by splitting the read into different seeds, and matching these seeds with the indexing of the genome, using the *query* function. Of course, the size of the seeds should be equal to the size of the indexed k-mers of the genome.
- For each seed, we now fill and store the information on where it matched with the indexed genome.

- Now that the seed part is done, we start the extend part. This part is launched by the *extend()* function. This function will loop over the different reads that we have. For each read, it will call another function which is *find_best_candidates()*.
- As discussed previously, this function will take all the seeds of the specified read, and output the best seeds.
- Here is how we determine the best seeds : for each seed contained in a specific read, we align this seed to the k-mer in the genome. Then using the function *extend_seed* we extend the seed by adding 1 nucleotide to the right, and another one to the left by looking at the reading this seed was generated from. We add nucleotides to the left and right side of the corresponding k-mer of the genome also, but taking the corresponding nucleotides from the genome. By adding both nucleotides we now compare the extended seed, to the extended k-mer and compute a score based on the numbers of nucleotide that didn't match. The higher the number of unmatched nucleotides, the lower the score.
- From this we get a score for each seed of each read. We use the function *find_best_candidates()* to select the most promising seeds of each read.
- To do that, (and it's something I adopted because I taught it was logic and it also worked well), we take the first 5% of the seeds with the lowest score. Then we remove all the seeds who have a score lower than -3 . Which means we remove the seeds that have more than -3 mismatches (discussed later).
- The output of this entire process is a dictionary with each read, its best seeds as well as their position in the read and the position of the corresponding k-mer in the genome.

3 Variable Selection

As you have seen above, at many times in the implementation I had to make a choice on the selection of a certain variable value. Here are the major values I chose, and why :

- Size of k-mers : varying the size of the k-mers will have an effect on the entire implementation. For example, if we choose a small k-mers size (considering the splitting interval is constant), then we'll have a bigger number of splits in terms of the

genome. We'll also have more seeds for each read. On the other hand, the k-mers of the genome will be smaller in size, and so easier to store. And, we shouldn't forget that if the k-mers are small, it means it will be easier to match them, so the index table of the genome would probably have a smaller amount of rows. If we have a bigger size of k-mers, well the number of seeds will diminish, as well as the number of splits for the genome. But as said previously, if a k-mer is of high length, then the different possible k-mers available in the genome are bigger, then we'll probably have a bigger index table. In my case I went for seeds/k-mers of size 10. I tested many values and realised 10 had a good result in terms of efficiency/quality of results.

- Size of the splitting interval : this variable represents at which frequency should we take seeds from a read, or k-mers from a genome. If it set to one, and let's say the size of the k-mers is 5, we would take 5 nucleotides every nucleotide. I decided to set this value to 1 for better results. For the reduced dataset it worked, but for the big dataset, too much RAM was demanded
- How to choose the best seeds : like I previously said, I decided to choose which seeds were the best by using the match and extend method. I decided to remove a score of -1 for each error because for me, it seemed like a linear loss would be good enough. I also took 5% of the best seeds because I realised that too many seeds were chosen, and I need to take the best. After doing some tests, 5 was the value that got me the right amount of seed, to not have too much nor too little. I also set the error margin to -3 , after running some tests I noticed that a lot of seeds would get chosen even with an error margin of -9 or -10 which is quite a lot knowing that the read is of length 36. So I decided to go for -3 but this highly depends on the length you choose to give to the seed. Since I was using a length 10, -3 was a good option and it was giving good results.

4 Results and Conclusions

By running my implementation on the reduced dataset, I obtained the following result :

- 50.16 % of reads were mapped

By running my implementation on the complete dataset, I obtained the following result :

- 64.18 % of reads were mapped

Please note that the accuracy of the previous result, for the complete dataset could be highly prone to errors due to the lack of RAM in my computer.

As a conclusion, I can say that through this project I learned more about read mapping and its different approaches. Many ways of improving my implementation are possible. Global alignment between the extended seed and k-mers as well as memory optimisation methods could be totally used to improve the performance of my algorithm.