76553 | Computational Genomics | Exercise 1

Guy Lutsker 207029448

Question 1

Preface

In this exercise we experimented with trying to map short DNA reads to a genome. Specifically we tried mapping two read files **ATAC.chr19.R01.fastq.gz, ATAC.chr19.R02.fastq.gz** containing 677996 reads each, to human chromosome 19. The general heuristic presented (one of several actually) in class suggested using hash tables to perform the mapping. This idea, as ingenious as it sounds at first glance, reveals some problems once you try to use it. In more detail, using hash maps will provide some useful results, but it cannot handle the slightest of variations to the reference genome, and so if our reads has a mutation / unexpected SNP / sequencing error our hash function will fail (by our definition). Taking this into account, I tried to design a genomic mapper that will map as many reads as possible, keeping in mind to try and conserve time and space complexity.

My Heuristic

My first idea was inspired by something said in the lecture (as one might expect?) - that DNA sequences longer than 20bp should be pretty much unique. And so I tried using a seed (K) of size 20 to hash every sequence of length 20 into a python dictionary, and trying to use the first 20 bases of each read to try and map it. This idea worked, since almost all hash entries were unique, but unfortunately the success rate (reads mapped) was ~50%. In addition when I remembered that we can try and refer to each read as both itself and as its reverse compliment did not boost up the success rate. This was mainly due to the problem I explained previously - hash of this length was unable to capture the diversity of the read variety. A naive move to the opposite end of the spectrum and to try and use a short hash length, (say $K = 5$) would result in having reads mapped to too many incidences along the genome. To be honest I tried many methods to try and solve this problem, including (but defiantly not limited to) trying to use a short K multiple times along each read, and try and merge these results to a likely candidate - this method was computationally not viable (at least I wasn't able to find an efficient implementation), although I still feel that it might have the highest success rate. One important thing I found is that the length that sequences start being unique is ~20 and the length of the reads is 36/40 which is almost double. Eventually I settled on this algorithm:

My Algorithm:

1. Set $K \sim \frac{Read\ Len}{2}$
2. Generate hash map **genome_map** from genome with key of length K.
   (**genome_map** is implemented using open hashing, with every key being a DNA sequence of length K and every value being a python set object.)
3. Set **map1** to be the translated version of all reads of index $0 : K$
4. Set **map2** to be the translated version of all reads of index $Read\ Len - K : Read\ Len$.
5. Set **Union** as the pairwise union of **map1**,**map2**
   (Now, for the i'th entry of Union I have the set of indices that match genome[i: i+K] at Union[i].)
6. For each indices in **Union**:
   (a) If len(indices) = 1: We are good.
   (b) If len(indices) = 2: Check pairwise match of both indices against the whole read (and not just sequence of length K). Remove the one with the lower score.
7. Do steps 1-6 on reverse compliment of each read to get **Union_RC**
8. Set **Super_Union** as the pairwise union of **Union** & **Union_RC**
9. Run step 6 on **Super_Union**
10. Return **Super_Union**

Question 2

In this section I will share my empirical results.
Firstly Let us look at the success rate of our algorithm for different K values:

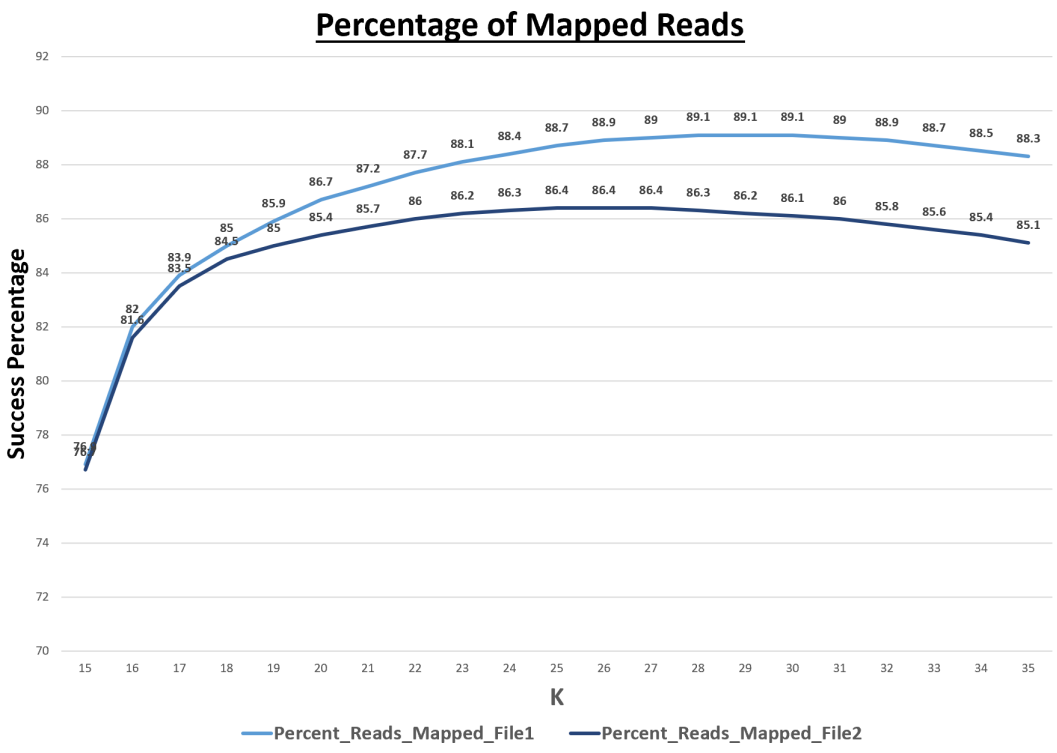

Figure 1: Percentage of Mapped Reads Per K

As we can see File1 has consistently higher success rate than File2, I attribute this result to the fact that the reads in File1 are longer (40bs) than File2 reads (36bp) and so we had more information to match. Altogether at its best the algorithm was able to map 89.1 & 86.3 percent of the reads in files 1&2 respectively ($K = 28$). We can also see that the results resembles a distorted parabola, this is intuitive since, we would expect a low success rate at very low or high K values, and would expect to find a sweet spot some where in the middle.

Another way of visualizing my results is using a histogram of mapped reads:

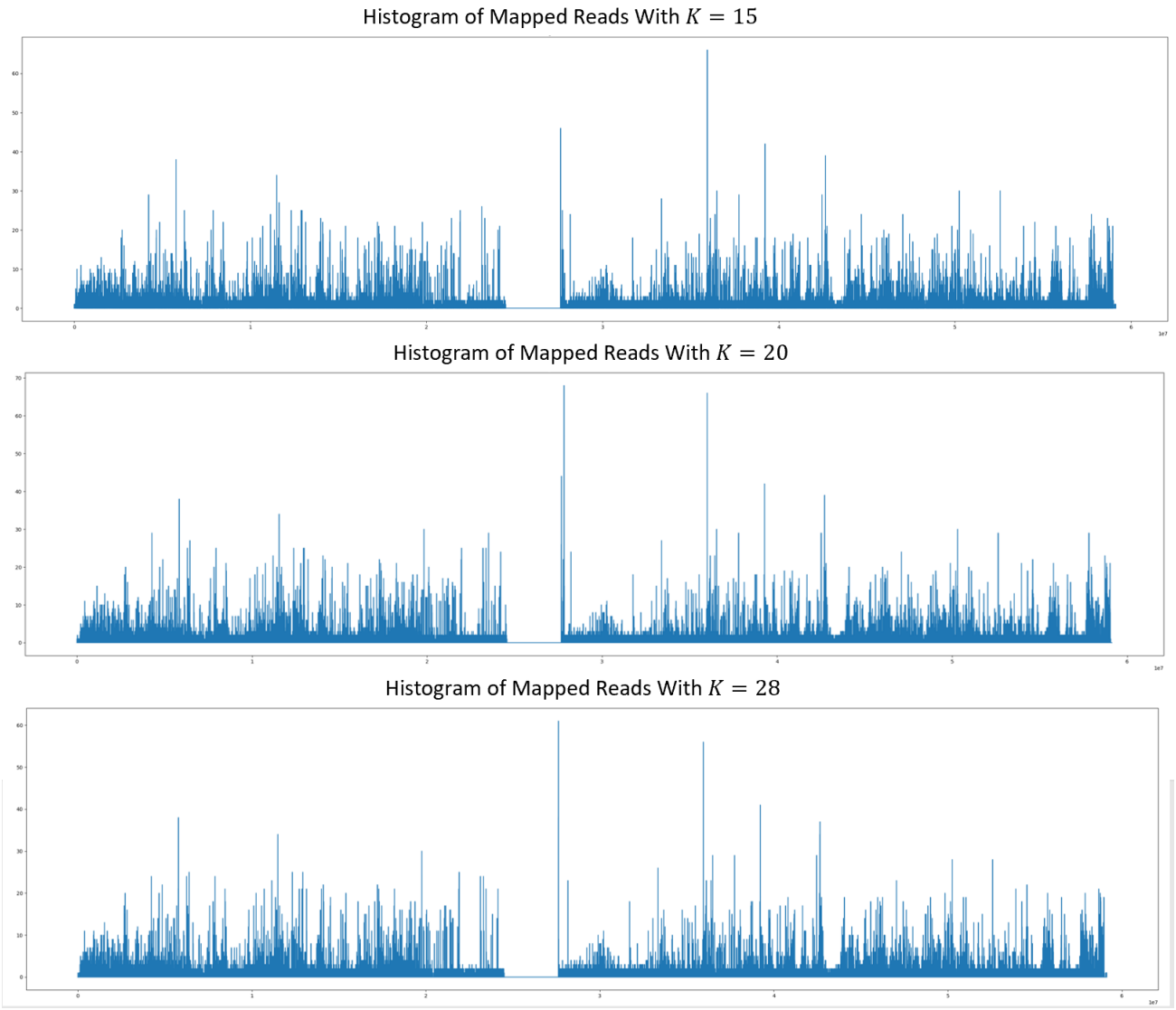

Figure 2: Histogram of Mapped Reads Per Certain Values K

I chose these values of K since 15 was the lowest K I tested, and it seemed interesting to me to see how it mapped, 20 because it was my initial naive guess for best results ($K \sim \frac{Read\ Len}{2}$), and 28 was the actual best result. As we can see the graphs are actually quite similar, but one quirk they all have is this gap in the middle of the histogram. When I first saw this I thought this might be a section that wasn't sequenced, and when I tried looking in the data itself (since I was a bit scared this was a bug) I saw that this was a sectioned filled with N values. And then I thought that this might be a chromatin packed area that is hard to sequence. Maybe I'm way off but since this section is in the middle of the chromosome this is the centromere of the chromosome? it would fit my results and would coincide with biological intuition.

Another Important result we want to measure in the time complexity of the algorithm. I wanted to split this analysis into two: (a) Analysis of time complexity of creating the hash table. (b) Analysis of time complexity of the algorithm itself. Here we can see the results of the practical time measurement of the algorithm itself:

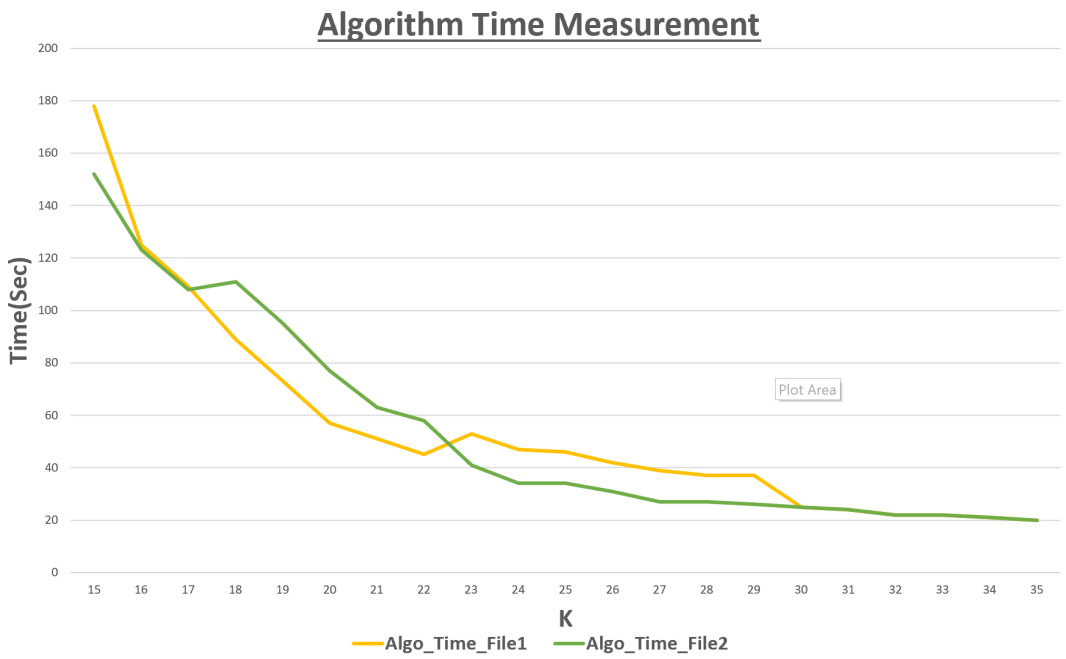

Figure 3: Algorithm Time Measurement Per K

As we can see as K grows larger, the time required for the algorithm decreases. This is also intuitive since my algorithm benefits mostly from the sparsity of the hash table entries, and so as K grows larger we would expect the sets at each value to be shorter and so my algorithm would have less work to do.

As I will show in figure.5 the space complexity of the algorithm heavily depends on the value of K. This is because K has a big impact on the size of the hash map, and this is the main parameter that takes up space in this program. To combat this I realized that in real life and in practice, we already have our reference genome laying around and we get all sorts of reads we want to map. And so its inefficient to create our hash map each time we want to map reads, since this step is interdependent of the reads themselves. This is why I wanted to plot the time it takes us to create the hash map versus the time it takes to pull it from memory if we already calculated it (this is basically why I wanted to split the time measurement into 2 parts), results are in the figure below:

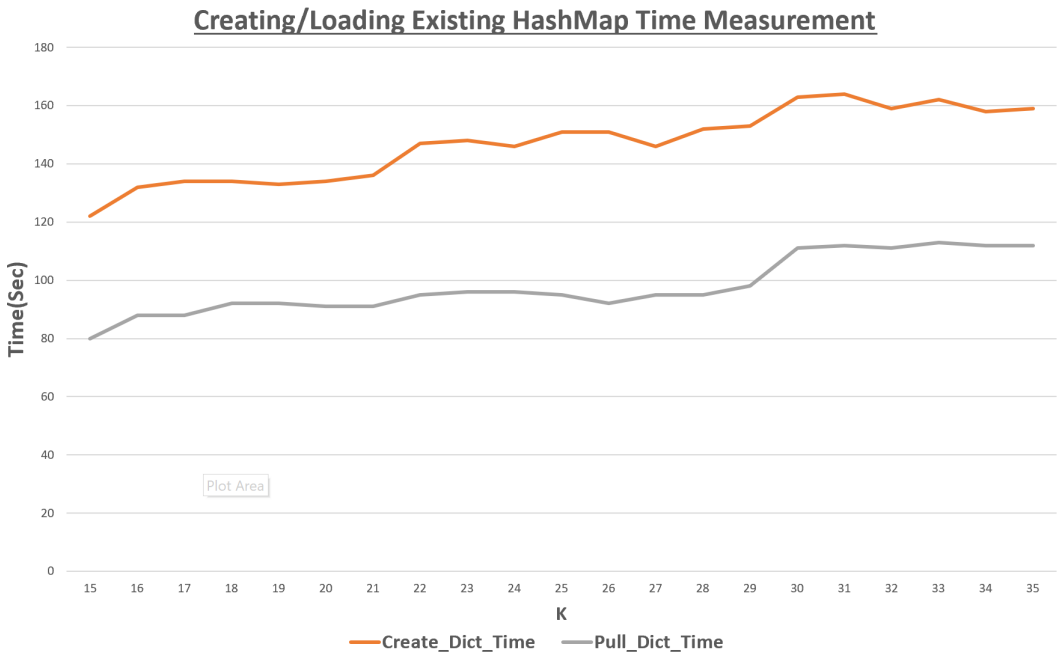

Figure 4: Time to Create Hash Map V.S Loading From Memory Per K

As we can see the time it takes to create/load the hash map goes up ever so slightly with the rise in K. What is important in this graph is that there is a consistent gap (of about 40 secs) between creation times and loading times, which can save a lot of time and money in practice.

And as promised, here I show the practical space complexity of the algorithm with relation to K. The results seemed too linear and I couldn't resist adding a linear regression fit :) The results are as follows:

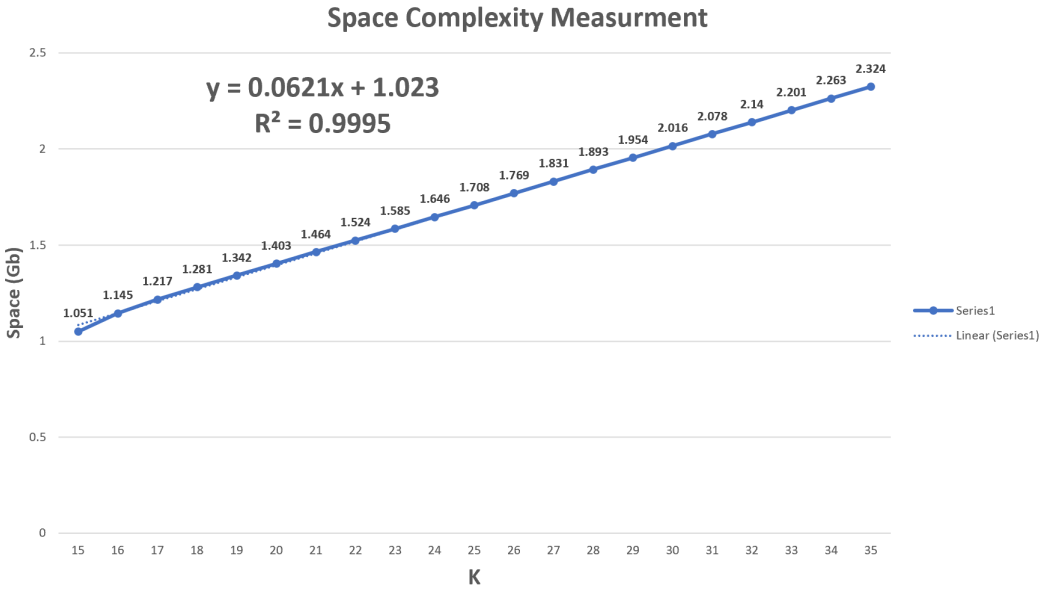

Figure 5: Space Complexity Per K

Question 3

No output required, will visualize results in Q4.

Question 4

When running the results from bowtie program in IGV I got the following results:

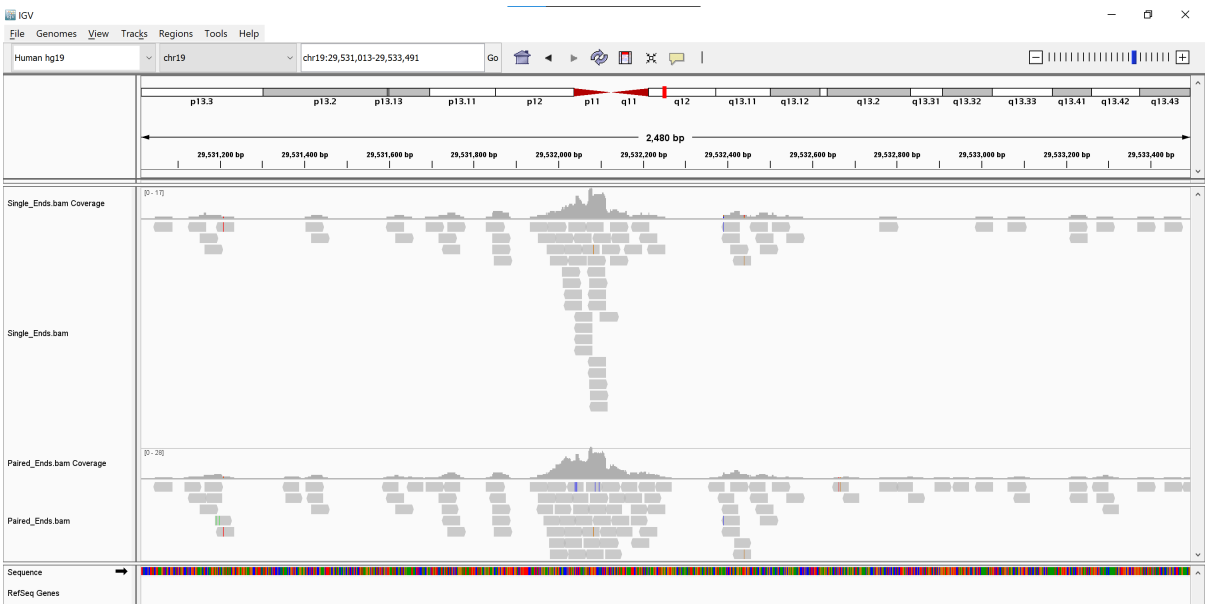

Figure 6: IGV Visualization

These results seem different than mine, and it looks as though their algorithm mostly found reads before (and a bit after) the gap I theorized was the centromere.