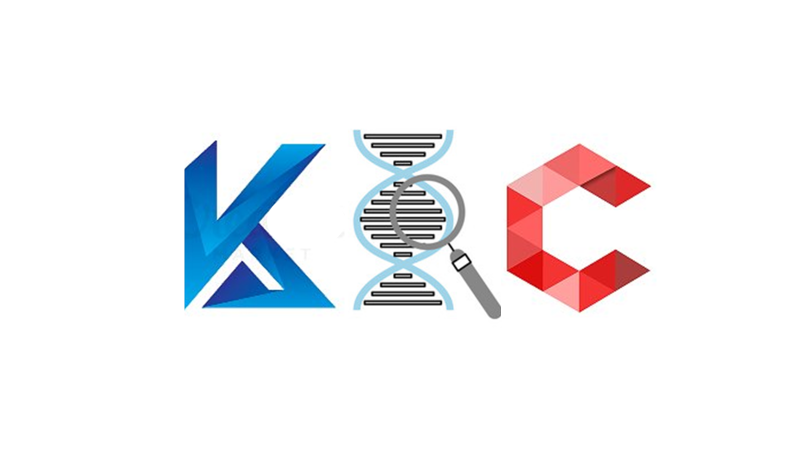
**kCompare:**

Fast and efficient comparison of metagenome sequences from patients

Byron Garcia, Ana Georgina Cobián Güemes, Genivaldo Silva, Rob Edwards, Daniel Cueves



EdwardsLab, Department of Computer Science, San Diego State University

ABSTRACT

Metagenomics DNA sequencing generates a vast amount of genome sequences that prior software can no longer analyze in a timely manner. Sequencing microbes by metagenomics is impacting all biological sciences, including medicine. We are using metagenomics to compare samples from patients over time, during the course of a disease. Over the last decade DNA sequencing has undergone monumental advances, but when it comes down to saving a life, every second is crucial. Therefore, we need faster, more efficient code to analyze DNA sequences and kCompare aims to accomplish just that.

**INTRODUCTION**

Cystic Fibrosis (CF) is a genetic disorder that causes a mucus build-up in the lungs. The mucus can act as a genome for bacterial microbes that have the potential of severely damaging or even killing the patient through lung infections. The recent advancements in microbiology has shifted the clinical CF research from the traditional petri dish culturing to entire genome extractions. Though this practice provides much more substantial data, it takes a long time to process it. The current amount of time to complete this process takes, on average, around 36 hours.

kCompare is an analytical tool, written in the C programming language, with the goal of generating smaller, more precise data to shave off time from the analytical portion of the process. We opted for the C language because it’s a lower level language that compiles and executes faster than higher, more abstracted languages like python or java. The idea is to add an extra pipeline right between the steps of going from read classification to analysis/taxonomy. We want to instead compare two separate instances of read classifications, which are formatted in FASTA text files, from two different genome extractions. The first file is supposed to resemble a period when a patient is healthy while the second is when the patient is not. From this point forward let’s define these as file1 and file2. The comparison is done using KMERS which are every substring from a read, iterated from left to right. An example of breaking this down to 2-MERS: ACGT 🡪 AC CG GT. We break down the two FASTA files into KMERS and compare those to each other as opposed to the entire FASTA files. The goal when comparing the KMERS is to locate unique KMERS that only reside in file2 and nowhere in file1. After we have the unique KMERS, we go back to file2 and extract entire reads that contain the uniques into a separate FASTA file. In turn, we exit the pipeline with a much smaller FASTA file that’ll likely contain the bacteria microbes and we put that into the analysis phase of the process rather than coursing the entire file2 through analysis.

The project’s functionality is broken down into three major components: Obtaining KMERS, finding the unique KMERS, and then locating the uniques within the file2 FASTA to create a new FASTA. Moving forward. I’ll explain more into depth how these components were solved and the efficiency of the algorithms from a computer scientist’s perspective. This is done with [Big 0](https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/) notation which describes computer algorithms as arbitrary functions, categorized by growth and running time with respect to inputs.

**OBTAINING K-MERS**

The files we worked with were reads directly from Ana Cobian’s Cystic Fibrosis Rapid Response project. These can be seen towards the end of the paper (**figure 1**). The specific files are from days 8 and 7 counted down to the patients passing. In this situation the day 8 represents our file1 and day 7 our file2. The sizes of file1 is 122MB and file2 is 113MB. We utilized another software tool called [JELLYFISH](http://www.cbcb.umd.edu/software/jellyfish/), version number 2.2.0, to get our KMERS. KMERS are used widely throughout the field of bioinformatics for different lengths entailing different purposes. For this situation, we use a small K-MER length because they are more efficient compared to entire reads when applied to data structures. The JELLYFISH executable takes two steps: first imputing the files along with the parameters of KMER size and initial hash size which outputs a binary file, and then imputing the binary file into a dump function that formats the K-MER files into readable text. I tested JELLYFISH out with different lengths to see which would be best. I broke down file2 into 11, 12, 13, and 14 mers. I couldn’t go lower than 11 for unknown reasons and stopped at 14 because the resulting kmer file was 134MB, 20MB more than the original file. After consulting with Ana, we decided to push forward with 11-mers because the file size was 40MB.

The way JELLYFISH was directly implemented into this project was through two C functions, SYSTEM() and SPRINTF(). SYSTEM() takes in a character array and executes it as if it’s a terminal command. SPRINTF() creates dynamic character arrays through manipulations based on inputs. The two of these together with command line arguments allowed me to virtually stack the JELLYFISH subroutine on top of my project. After figuring this out, I added slight multi-threading that would split off two processes to execute JELLYFISH on two of the files at the same time in lieu of executing one after the other. While the JELLYFISH process doesn’t take a terribly long time in this exact situation, dealing with much larger FASTA files and K-MER lengths, then concurrently running two processes can cut a large run time in half. When this component is broken down into complexity notion, without multi-processing it would look like 0(n + m) where n and m are the two jellyfish subroutines under a single function. If the two files are similar sizes, it would look more like 0(2n) but after splitting it into two would simply be 0(n).

Oddly enough, this component of the project is the newest. Prior to this, I had used pre-set files where the FASTA had already been broken down into KMERS. Due to this, I had largely used JELLFISH as a black box and only toyed around with its functionalities by analyzing inputs and outputs. The next step to progress this component would be to better understand how JELLYFISH works and maximize its utilities. The largest problem I have here is determining the best input for initial hash table size.

**UNIQUE K-MERS**

Detecting the Unique K-MERS was actually the first component we worked on. When given the problem, we went to the drawing board and brainstormed ideas. At first glance we looked at simply going through the each file line by line and just comparing them but that’d result in a worste case complexity close to 0(n\*m) where each would be the length of both files. This would be the case because for each KMER in file2, we would have to compare it to every KMER in file1. We eventually opted for a hashtable because that’ll be 0(n+m), where we would only have to look at each element from both files only once.

Multiple layers of structs were implemented for the sake of readability: struct hashtable -> struct node -> struct kmer, where the kmer is a character array, the node consists of a kmer struct, and the hashtable is an array of nodes. The simple methods followed: Create Node, Add Node, Remove node, and Find node. All of these are used in later functions but never utilized directly in the driver.

The component goes as follows: Open up the KMER file from file2 and count how many lines reside then multiply that by 1.5 to get a solid hashtable size. Go through the file line by line, grabbing each KMER, hashing it to get an index, and placing it into the hashtable data structure. Once we go through all of the KMERS in file2, we close it and head to the file1 KMERS and go through a similar process. We go through line by line and apply the same hashfunction to it, but instead of adding it to the hash table, we check the index it created and see if the same k-mer is already there. If it is, we remove it from the data structure and if it’s not found, we simply pass over it and check the next KMERS. In the end, our data structure will consists purely of KMERS that were only generated from file2 and were no where within file1.

We actually went through two hashtable functions. The first was rather dull, a simple quadradic function that was temporarily created for the sake of completeness. We didn’t actually come back to it after completing the next component. We created a simple method that would go through and count how many nodes were in each index of the array to determine the collision rate and knew it had to be changed. We came across a function called [djb2](http://www.cse.yorku.ca/~oz/hash.html) which used bit shifting to a great extent, resulting in far less collosions and pushing the search closer to the complexity of 0(1).

In terms of logic, this portion of the code is solid. The issues would lie in the coding itself. We ended up writing the data structures by scratch in place of a built-in library for me to get a better understanding of the C language. There would probably be better results if we replace the scratch code, but we decided to push through to the next stages and worked with what we had because we had already built on top of it.

**EXTRACTING READS WITH UNIQUE K-MERS**

Once we have the table of unique KMERS that are purely from file2, we need to go back into file2 to grab the reads containing them. Of all the changes each component has undergone, this one was by far the most drastic when taking performance under account. Again, we worked with dummy values and already had the KMERS from them.

Our first iteration to this component had this format: We applied the STRSTR(str1,str2) function which would return true or false depending on if one input was a substring to the next. We open file2 and look through line by line, and for each read we applied the function for KMER in the table. If we hit true at any point, we copied the read and streamed it into another output file, then moved onto the next read. This resulted in a complexity of 0(n \* m) where n is the number of lines in file1 and m was the number of KMERS in our table. After completing this we pushed to make it faster by applying POSIX Threads which sped up the process by X3. We were ecstatic that we finally got this all working, but it turns out this was one of the most naïve way to solve this problem.

Preliminary results of our first iteration with 3.36MB healthy 9-mers and 2.65MB of sick 9-mers would course through FASTA file of 250MB in around 30 minutes.

Our second iteration took a very different approach to searching through the table. We coursed through each read with a nested for loop and accessed the KMERS of the reads. Within each iteration of the nested loop, we apply the hash function to the given KMER to see if it shows up in the table similarly to how we found the unique KMERS. If the KMER from the read isn’t found, we iterate over one character in the read and apply the function to that KMER. If we find a hit, we just copy down the read and output it to a new FASTA file and move onto the next read. This was the initial algorithm presented but it sparked controversy because of the nested for loop. After telling the project leader, he didn’t like method of using the nested for loop, but when telling the lab instructor, he simply told me to test it.

Preliminary results of our second iteration with the same files computed and performed far better. When testing the same KMER and FASTA files from our first iteration, it computed in about 49 seconds! At this point we knew we hit a good spot. Shortly after the poor hash function was replaced, I contacted Ana Cobian and I introduced to her the project I was working on. From there forth, we collaborated to see how the project can best benefit hers. It was time to test the software on more realistic data.

**RESULTS**

In terms of functionality, the kCompare software works exactly as intended:

argv[0] argv[1] argv[2] argv[3]

. /kCompare file1 file2 k-Length 🡪 output

Output = file1-VS-file2-{K}MERS.fasta

It takes in two FASTA files, breaks them both down into KMERS, locates the unique KMERS, goes back to file2 that has the unique KMERS, then creates a new FASTA file of reads that hosted these new KMERS. I tested the software with the data provided by Ana, which are shown in **figure 1** below. These files are days counting down to the passing of the patient, the days: 303 297, 8, and 7. We did 303vs297, 297vs8, and 8vs7 in both directions. The sizes of the inputs, outputs, and execution time are all provided in **figure 2**. The initial idea was to compare the output sizes of one to another and justify them. 303vs297 are similar sizes but several days away from eachother, so it should generate a large file relative to the originals. When comparing 8vs7, this statement holds. Despite 8vs7 having much larger files, the generated files are smaller by ratio. When looking at 8vs297, the outputs were obvious. The larger file in position file2 generated a huge file while vise-versa file1 generated a much smaller one.

The file sizes were a basis of what we were looking for before moving forward. However, we knew we had to dive further. Ana created an awesome BASH script that executes [blastN](https://www.ncbi.nlm.nih.gov/Class/MLACourse/Modules/BLAST/nucleotide_blast.html), a program that does a nucleotide query search on a FASTA file. In short, it tells us what micro-organisms are represented by the reads. We first did entire searches through day8 and day7. Then we did a search on the day8VSday7 FASTA file to see what micro-organisms the software detected between the two. What we expect the reads in the VS file to show are the largest difference or shifts of organisms from day 8 to day 7. The largest spikes should be the top hits on the VS. If we look at **figure 3** and **figure 4**, the results are mixed.

The game plan Ana and I came up with was to take the blastN results from days 7 and 8, and then subtract the difference. The hit counts that show high numbers in day 7 that weren’t present in day 8 should portray a spike of bacterial activity. From there we create virtual rankings on what compare results should be. We take the top 15 hits from day 7 and find the same bacteria name in day 8 and subtract that value for all of them. This is all shown in **figure 4**.

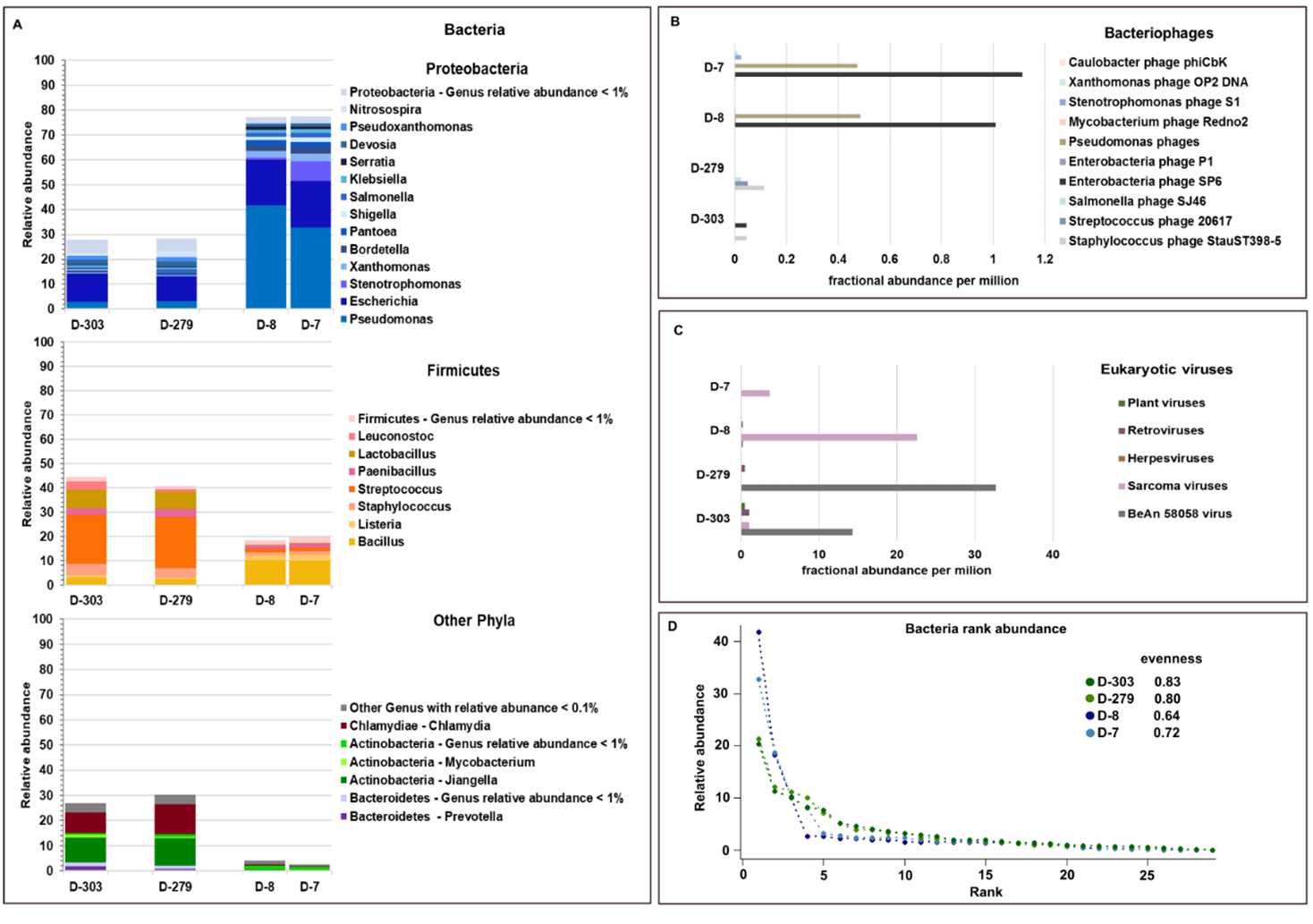
In **figure 5** we show the actual results of day8VSday7-11MERS.FASTA from blastN compared to our virtual rankings. Because Pseudomonas showed a large negative number, we didn’t think anything from day 7 would show any unique values of that bacteria. The rankings after that were very close! The largest spike was present on the top values of the kCompare file, and the rest were either shifted by one or two places. This is perfect since the day8VSday7 file is only 15MB versus the 115MB of day 7, yet the top hits remain close to the same. The last thing to do in this situation would be to record the amount of time it would take.

**Conclusion/Future**

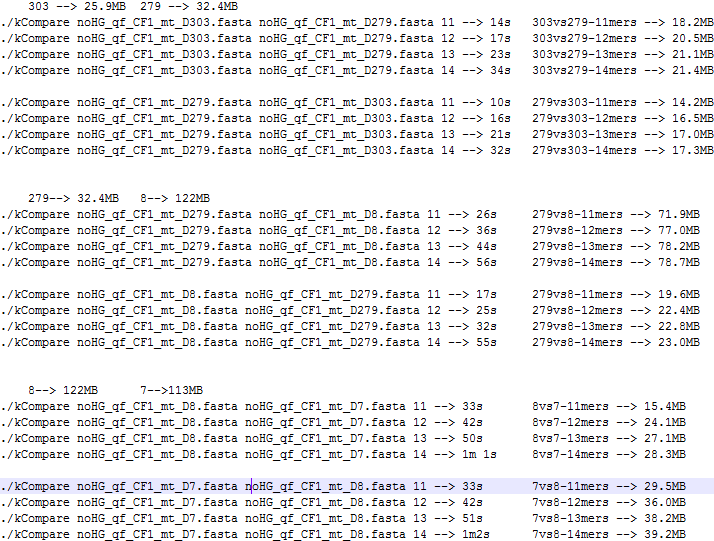
Through several hours of trial and error, classes and rewrites, I ended here. The first major issue came from hardcoding several of the parameters from the dummy values previously given. Most re-writes were implemented to work with more dynamic inputs and different files along with all the classes that taught more fundamentals that dove deeper into the C language. When speaking to one of the lab’s prior leading figures, Daniel Cuevas, he stated this project can be used for any situation regarding two genomes broken into KMERS. This warmed my heart, because if this application towards cystic fibrosis were to fall through, the software could find some use else-where.

As for the research towards cystic fibrosis, plenty more blastN trials need to be conducted. More FASTA files along with different K-MER sizes need to be tested. On top of that, the amount of time the blastN needs to be recorded to really hit the goal of creating a faster project. It’s a given that a 15MB file would run faster than the 115MB, but how much of a difference would be great to clarify. It was built into the script Ana constructed for me, but it bugged out. Also, a better simulation of the healthy and sick files would be better for the sake of testing. That being said, I’ll be more than happy to keep in touch with Ana and continue working on this project to really reach its full potential.

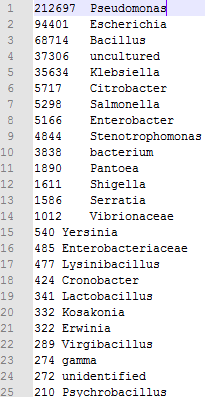
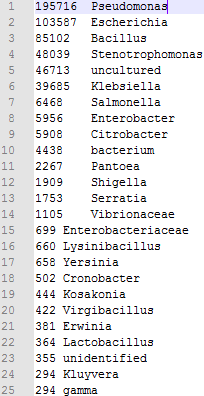
When comparing KMERS to eachother it is more efficient but there is a point that the generated KMER file size surpasses that of the original FASTA file. This raises the question of how comparing the FASTA files one-to-one would perform. I understand researches are curious of comparing KMERS but I feel the accuracy from one-to-one would be beneficial to the rapid response project and cystic fibrosis research. This could be a potential project for future undergrads to take on.

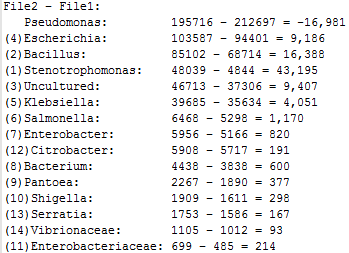


**Figure 1:** FourBacterial communities extracted from patient CF01. (Ana’s Rapid Response)

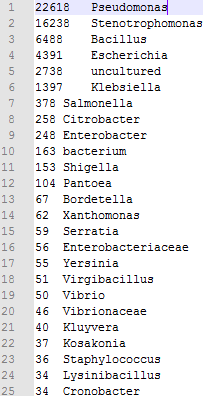
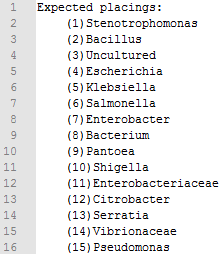


**Figure 2**: Execution examples, comparison of days in both directions.



**Figure 3:** The top two portray the blastN results of days 8 and 7. On the bottom shows the subtraction of day 8’s values from day 7’s top 15 hits to represent the spikes. The numbers to the left show the ranking of the spikes.



**Figure 5:** The left shows the predicted rankings based on the spikes from day 8 to 7 results from blastN . The right shows the actual blastN results and where they fall.