Matching Sequencing Reads To Contaminants

EN. 600.439 Computational Genomics
Final Write Up

Members:
Yu-chi (Kiki) Chang
Jeana Yee
Mariya Kazachkova
Min Geol Joo (Kevin)

Abstract:
In biology laboratories, contamination of samples is a common issue that has been difficult to resolve, for contaminants are widespread and their small size makes them hard to detect. With the development of high-throughput life science instruments such as next-generation sequencing, scientists can now detect contaminants in the sample of interest by searching for contaminants' genome in the sample's sequencing data. However, the large-scale sequencing data requires fast searching and efficient memory storage. In addition, sequencing errors due to natural variations as well as instrumental error need to be resolved. Here in this project, we developed a program using a combination of an FM Index, the pigeonhole principle, and the naive algorithm for approximate matching in attempt to provide an efficient way to align reads to a reference genome allowing 2 mismatches. Our result suggests that our algorithm is correct (in that no exact, 1 mismatch, and 2 mismatch cases are missed), but because of the need to store each genome even though an FM Index is being built over it (discussed later) we are hesitant to declare it as an efficient option if one is considered about space. We hope to extend upon this, allowing k-mismatches.

Introduction:
After reading various articles **[1][2],** it became clear that the presence of contaminants is a problem often faced by those collecting data sets of sequencing reads. We found this problem engaging as it is actually two problems wrapped up into one: there is the problem of an efficient algorithm/data structure that can be used to match reads that contain contaminants with their respective genomes, as well as the problem of knowing which contaminants' genomes to match against. Thus, we approached this problem in two separate parts.

We tackled the latter problem mentioned above by collecting/reading numerous articles that discussed the presence of various contaminants in the data sets examined by the particular article. We were then able to compile of list of contaminants that we believe have the highest chance of being present as a contaminant in a set of sequencing reads. Our original list was composed of the following: human, dog, cat, mouse, rat, three kinds of mycoplasma (*Mycoplasma hominis, Mycoplasma hyorhinis, Mycoplasma fermentan*), one kind of Acholeplasma (*Acholeplasma laidlawii*), Staphylococcus epidermidis (found on human skin), and pine (commonly found in dirt)*. We chose to include human because there is a chance that the researcher's DNA could get into the sequencing reads while performing the experiment, and we included dog and cat because these are species that humans (specifically the researchers running the experiment) could have been in close contact with. We decided that mouse and rat would be good choices because these are species often found in labs (where the sequencing reads are being collected). From our reading it

became clear that both mycoplasma and acholeplasma are common contaminants present in data sets, so we decided to include those as well. Furthermore, we also chose to include a bacteria found on human skin, as well as pine (due to the possibility of the presence of dirt on skin).

Once this list was compiled (and all of the genomes were collected and placed into a directory) we tackled the original problem: creating the algorithm to efficiently match the reads in our data set with the set of genomes we had for our contaminants. We began looking at articles that discussed various matching algorithms, and we came across one that struck us as interesting. This article**[3]** discussed using a bidirectional FM index to do approximate matching. However, the writing in the article was dense and no code was provided, making it difficult to judge the logic and validity of the article. Despite that, the idea of approximate matching using an FM index (specifically a bidirectional FM index) stuck and we began toying with algorithms that would utilize the index to do approximate matching in order to match reads with contaminants. After coming up with an algorithm and some attempts to translate the algorithm into Python code we stumbled across the realization that our idea had one major flaw that we had accidentally overlooked (this is discussed in depth in the methods and software section). After this, we started from scratch and ended up coming up with a different approach to doing approximate matching with an FM index. Our new algorithm (discussed in the methods and software section) no longer needs a bidirectional FM index. Instead, it utilizes a regular FM index along with numerous concepts used in class (pigeonhole principle, naive algorithm and benefit of not preprocessing p) to use approximate matching to match sequencing reads from data sets with the genomes present in our directory of contaminants.

***Important Note on Introduction:**
Although we initially wanted to use all of these genomes, given time constraints and the fact that we wanted to do a thorough analysis of data we decided to only use the three Mycoplasma and one Acholeplasma. Additionally, because the data set of sequencing reads that we are matching is for the human genome, we decided against matching our reads against the human genome.

Prior Work:
<u>High Throughput Short Read Alignment via Bi-directional BWT</u>
Lam et al. **[3]** proposed a new indexing data structure called bi-directional BWT. The original BWT allowed for an efficient backwards search for exact matches, but was not sufficient for aligning genomic short reads because the best alignment of a read may contain differences. For efficient approximate pattern matching, they proposed a bi-directional BWT data structure that could support both forward and backward searching as well as switching from the forward to backward search or vice versa in the course of aligning the pattern.

For approximate matching allowing one mismatch, they start with a backwards search on the second half of the pattern, branch for every position in the first half to assume a mismatch at that position, and continue the backwards search until reaching the start of the pattern. A similar approach is used in the opposite direction to account for mismatches in the second half.

Their algorithm for a 1-mismatch alignment is as follows:

1. Partition P into P1 and P2

2. Backwards search on P2
3. For i = 0… len(P1)-1, where i is the position of the mismatch,
    a. Continue backwards search on P1[i+1:]
    b. Compute SA range of P1[i+1:]P2
    c. Continue backwards search on P1[0 : i-1]
    d. Report SA range of P
4. Forwards search on P1
5. For j = len(P1)… len(P2)-1, where j is the position of the mismatch,
    a. Continue forwards search on P2[:j-1]
    b. Compute SA range of P1P2[:j-1]
    c. Continue forwards search on P2[j+1:]
    d. Report SA range of P

BatMis

Rather than using a BWT branching approach, BatMis (Basic Alignment Tool for Mismatches) **[4]** is a BWT based aligner that uses a seed and extend approach, and is specialized for short read mapping.

First, P is partitioned into two halves. Substrings of reference genome T that have up to k/2 mismatches with P1 or P2 are found by recursion. This is followed by recursive prefix extensions until the length of the substrings are equal to the length of P and the number of mismatches is less than k, or the difference between the substring and the same length substring of P is = k + 1 mismatches. A similar process is done with recursive suffix extensions. The list of strings generated by prefix extension and suffix extension are used to compute all k-mismatch patterns of P to T by unioning the two lists. BWT is used to obtain SA range (rows in the suffix array with P as a prefix), and a condensed suffix array is used to decode corresponding locations in T. The BatMis algorithm is able to be used for large number of mismatches, since it avoids having to branch.

Kraken

Kraken **[5]** is a sequence classification tool that utilizes exact alignments of k-mers and a novel classification algorithm. It is based around a database that contains records consisting of a *k*-mer and the lowest common ancestor (LCA) of organisms with that k-mer in their genomes. This database allows quick lookup of nodes in the taxonomic tree associated with a k-mer. We will run our genome and data using miniKraken (with smaller database) and compare the result with our program.

Materials:

FM Index obtained from Ben Langmead's GitHub notebooks **[6]**

For real sequencing reads to test against our software, we used GEO to download a cancer experiment miRNA sequencing dataset (SRR3279553). These reads were then processed to take out any reads that had sequencing errors, and were clipped using the BBDuk software and the Illumina 3' adapter sequence specified in the kit that was used by the experimentalists.

Methods and Software:

We created a software that, given a data set of sequencing reads, scans for common contaminants. Given the length of the contaminant genomes compared to the length of the reads, we decided to take an offline method approach that builds an FM index for each one of the contaminants. As discussed in the introduction, we spent a large amount of time trying to implement one algorithm before realizing there was a flaw in it and moving on to what ended up being the algorithm that we stuck with. Both of these algorithms will be discussed in this section.

Bidirectional FM Index Method

Why FM Index:

We had learned in class that the FM Index was an efficient option for doing exact matching since the query time was linear and it did not require all of the text that the FM Index was being built over to be stored. Because of this, we looked into ways to incorporate an FM Index into our algorithm. We had learned how to use an FM Index for exact matching, so we thought looking into how to use this data structure to do approximate matching would be an interesting expansion on material covered in class (because, as we learned, there can be errors in sequencing reads, thus creating the need to do more than just exact matching).

Why Bidirectional:

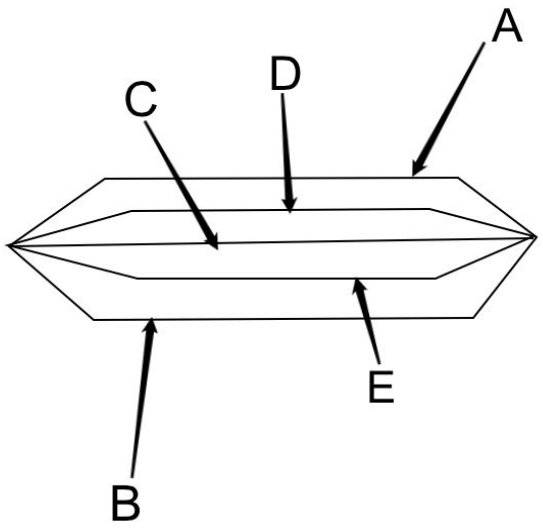
As stated in the introduction and in prior work, we got the idea for a bidirectional FM Index from one of the articles that we read. Our thought process behind using a regular FM Index and an FM Index built over a reversed version of the original text (genome) was so that we would be able to account for mismatches. The algorithm or finding approximate matches with up to one mismatch was as follows:

1. FmB = regular FM Index
2. FmF = forward-facing FM Index build over the reversed version of the genome
3. Match for P using FmB until cannot move further (so until we either find an exact match or we have gone as far as we can in P and have hit a mismatch).
4. If we have found an exact match
    a. Return that a match has been found
5. Store the location where the deepest mismatch occurred in P as a variable (let's call it deep_p_back) and store all of the locations where the deepest mismatch occurred in T as a list (let's call it deep_t_back)
6. Match for the reverse of P using FmF until cannot move further (in this case we must hit a mismatch because if it was an exact match FmB would have found it in line 3)
7. Store where the deepest mismatch occurred in P as a variable (let's call it deep_p_forward) and store all of the locations where the deepest mismatch occurred as a list (let's call it deep_t_forward)
8. Transform deep_p_forward and the elements in deep_t_forward to correspond with what the index value would be if everything was backward-facing
9. If deep_p_forward = deep_p_backward this means that the mismatch occurred at the same place in P for both the forward and backward search
    a. If this is not the case move on to step 11

10. If deep_t_forward has any element in common with deep_t_backward (this means that that there is a mismatch that occurred in the same place in T for both searches)
    a. Return that an approximate match has been found
11. If we are at this step this means that no matches or approximate matches have been found. Repeat the process but start searching forwards (using FmF) instead of backwards.
12. Return the no matches or approximate matches have been found

We began to implement this algorithm. We got the forward facing and backward facing FM Indexes up and created the algorithm for reversing the indexes for the forward facing FM Index and reversed version of p before realizing that we could potentially have a large number of false negatives using this method.

The flaw that we found can be easily summarized by the following picture:



Let's pretend that each line in the above picture represents a sequencing read with one mismatch and each letter represents a possible place where that mismatch could be. After closer examination we found that our algorithm only accounted for the cases where the mismatch was at location A or location B; if the mismatch was between those two locations then our algorithm would not identify the read as having an approximate match. This is because of the "go as deep as possible until finding a mismatch" part of our algorithm. We are looking for the furthest mismatch, but it could be the case that a prior mismatch was the one that would have lead to an approximate match being found, and our algorithm did not take that into account. Because of this we modified our algorithm, thus leading into the next point in this section.

<u>FM Index and Pigeonhole Method</u>

While we liked the idea of using a bidirectional FM Index, we unfortunately were not able to come up with a practical algorithm that utilized it without branching, thus we got rid of the forward-facing FM Index and moved on to a different algorithm. By this point we were very set on coming up with an algorithm that did approximate matching using an FM Index, so we played around with various other

ways we could execute this. We finally settled on an idea that made use of numerous topics we learned in class (the FM Index, the pigeonhole principle, and the naive algorithm for approximate matching). We found that our idea was similar to BatMis in that it first exact matches a partition of the pattern, and extends to accept mismatches, so we decided to continue with our approach.

Our algorithm (for the one mismatch) is as follows:

1. Build and FM Index for the genome being matched to
2. For each sequencing read (p)
    a. Separate p into two equal parts, p1 and p2.
    b. Find all of the occurrences of p2 in the genome. Return this as a list
    c. For each index in the list
        i. Go to (that index - the length of p1) in the genome and run the naive algorithm (allowing up to one mismatch) on the genome and p1
        ii. If the naive algorithm is successful (one mismatch or less) append the index in the genome where you started the naive method to a list that will be returned
    d. Find all occurrences of p1 in the genome. Return this as a list
    e. For each index in that list
        i. Go to (that index + the length of p1) in the genome and run the naive algorithm (allowing up to one mismatch) on the genome and p2
        ii. If the naive algorithm is successful (one mismatch or less) append the index from the index list returned in step d to the list that will be returned
    f. Change the list into a set to ignore duplicates (would occur if something was an exact match)
    g. Return the set of matches

We also extended the algorithm to two mismatches. It follows the same structure as the algorithm given above except that p (each sequencing read) is separated into 3 equal parts, the FM Index is used to find exact matches of each one, and for each index in each of those three lists, use naive to approximate match the other 2 parts (with up to two mismatches). For each index in each of the three lists it is also important to count the total number of mismatches (since naive allows for up to two mismatches, meaning that if we exact match p2 we need to ensure that the total mismatches between p1 and p3 is less than or equal to two).

Brief Discussion on Naive:
We believe that the naive algorithm for approximate matching that we learned at the beginning of the semester is the best choice here. While we very briefly entertained the idea of Boyer-Moore, it does not make sense to use it because once we mismatch we can break; there is no need to slide or skip any characters. Also, doing naive allows us to not have to preprocess pi, which makes the verification O(n) using the naive approach.

Discussion on Time Complexity:
Let's say n is the length of p and k is the number of occurrences of p in the genome that we are matching against. Let's say m is the number of sequencing reads. For each sequencing read we must find k

occurrences of p1 and p2 (or p1, p2, and p3), which is O(n+k). Additionally, for each pi that we find exact occurrences of we will need to run naive on either half of p or two thirds of p (both ways is O(n)). This gives us O(nk), since k is the number of occurrences returned from the FM Index. Thus, our time complexity for a single read becomes O(n+k) + O(nk). Because we have m sequencing reads, our total time complexity is O{m((n+k) + (nk))}.

Discussion on Space:
While one is generally able to dispose of the genome after building an FM Index over it, because of the way we do approximate matching we must hold on to the genome. Because of this our time complexity is increased significantly, although it is still O(m) (for each genome. Since we have numerous genomes it becomes O(mc), where c in our number of genomes).

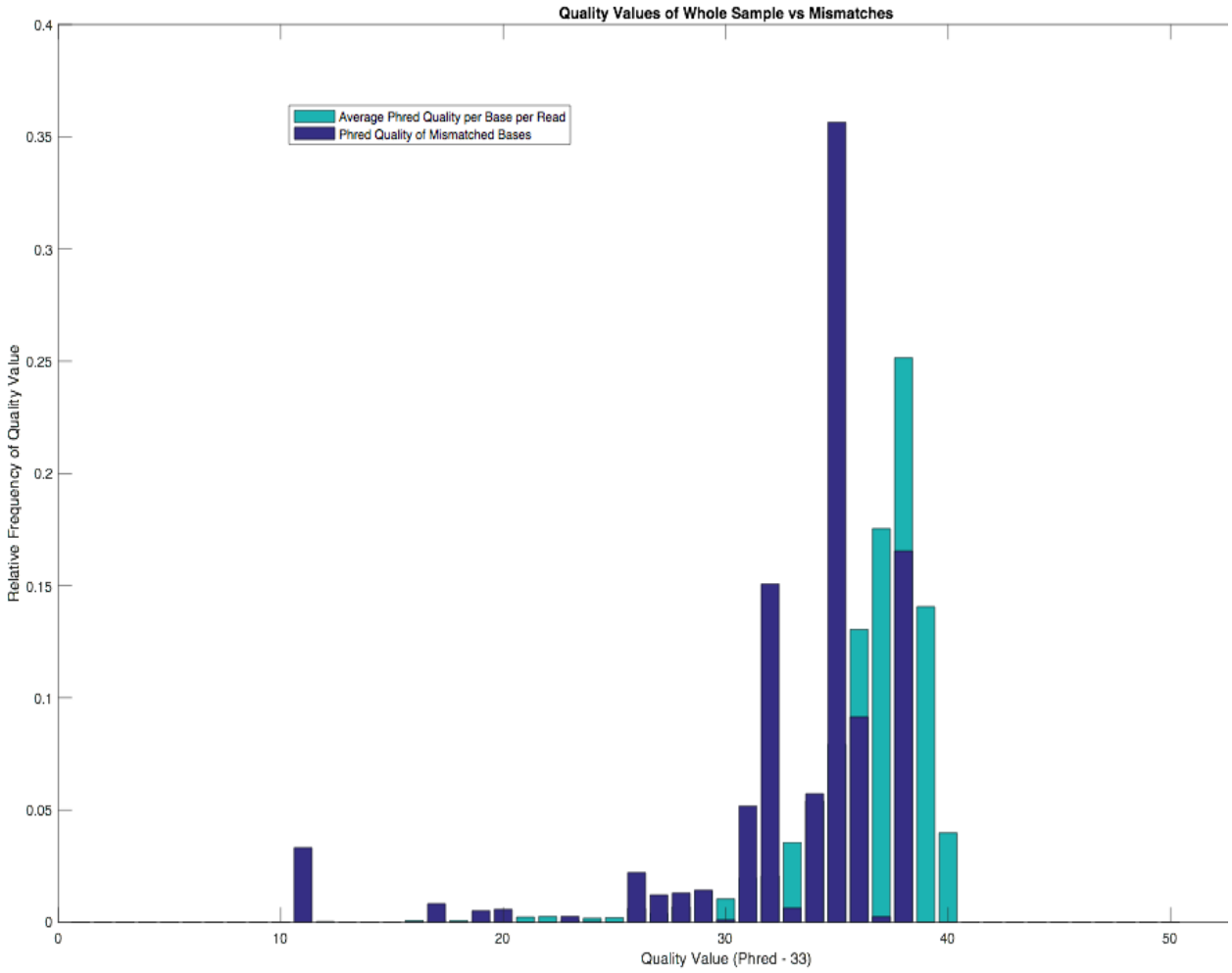
Discussion on Extending to k Mismatches:
If we wanted to extend to k mismatches we would need to partition each sequencing read into k+1 partitions. We would then need to do exact matching using the FM index for each one of these partitions and do the naive approximate matching algorithm for each of the k partitions that are not the partition currently being analyzed using the FM Index. Because the lengths of the reads that we are using can be as short as 5 nucleotides, partitioning each read into more than 3 parts quickly becomes impractical. Thus, although this process could be extended to account for more than two mismatches, we decided it would be best to stop at 2 mismatches.

***Important Note on Methods:**
Due to memory issues when running our software with the large genome files, we had to split the genome into parts of 40,000 bases. On average, this is one-twelfth of the entire genome.

Results:

Contaminant Detection Compared to Kraken
When running real sequencing experiment reads against substrings of 4 Mycoplasma species genomes, we obtained 274 unique hits, which matched to about 250,000 locations across the four genomes. Running the same file with the sequencing reads against the miniKraken database showed a lot of matches to viruses, but no mycoplasma, nor was there a mention of bacteria in general in the kraken report.

Quality
To check if quality makes an impact on mismatches, we looked at the quality scores for two different sets. The first set was the set of all reads that we used. For each read, we averaged the Phred scores, and used the average score for each read as data. The second set was the set of all reads that hit with one or two mismatches. For these reads, we took the Phred score of the bases that mismatched, and used these raw scores as data. We were able to obtain a frequency plot for each dataset by finding out how many times each quality score came up, and normalized the frequencies to the size of each dataset to get a relative frequency plot. This plot is shown below.



Although the difference in the distributions is not enough to show statistical significance, it is interesting to note that the mismatched bases have a left-shifted distribution. If this held true across entire genomes instead of fractions of them that were used, we could conclude that sequencing errors influenced the differences between amounts of exact, 1-mismatch, and 2-mismatch hits. It could also be possible that if there were less errors, Kraken would be able to identify Mycoplasma contaminant fragments at a higher threshold.

Conclusions:

Because we had to split up the genomes of the contaminants in order to run our program, it is difficult to make any strong conclusions about our results. We may have obtained no matches to Mycoplasma in miniKraken due to this setback, or it may have been possible that Mycoplasma is not in the database. Furthermore, the difference in distributions of quality values shows that the distribution of mismatch quality is generally lower in quality than the average.

One of the main things we learned from the assignment is that the FM Index data structure seems to be well suited for exact matching but becomes difficult to utilize when trying to do approximate matching. This is because of the nature of the algorithm, if a mismatch occurs some sort of branching must happen. In addition, there is also the possibility that an incorrect branch will be followed (for instance if our reads is “aaabcd” and in the genome there is “...cccbcd” and “...aaabbd”, the first branch will be followed but the second branch actually yields an exact match). We believe that although our solution is not the most efficient (we are forced to store the genome even though one generally does not need to do that while using an FM Index), it is able to avoid the problems stated above.

On a broader scope, one of the things we learned from choosing this particular project (that was only known to one member of our group because he has had exposure to genomic research) is the prevalence of contaminants in sets of sequencing reads. This was a new aspect of computational genomics (checking one’s sequencing reads to make sure that the reads come from the genome that is actually being sequenced and not some sort of genome that got in there by accident) that we had not considered before. Thus, we learned that there are factors besides quality values that can very influential in determining the quality of a read (if it matches to the genome of a common contaminant than it is likely not a quality read).

Literature Cited:

[1] William B Langdon. Mycoplasma contamination in the 1000 genomes project. BioData Mining, 7(1):1, 2014.

[2] Anthony Olarerin-George and John B Hogenesch. Assessing the prevalence of mycoplasma contamination in cell culture via a survey of ncbi’s rna-seq archive. Nucleic acids research, 43(5):2535–2542, 2015.

[3] Lam, T. W., Li, R., Tam, A., Wong, S., Wu, E., & Yiu, S. M. (2009, November). High Throughput Short Read Alignment via Bi-directional BWT. *2009 IEEE International Conference on Bioinformatics and Biomedicine,* 31-36. doi:10.1109/bibm.2009.42

[4] Tennakoon, C., Purbojati, R. W., & Sung, W. (2012, June 10). BatMis: A fast algorithm for k-mismatch mapping. *Bioinformatics, 28*(16), 2122-2128. doi:10.1093/bioinformatics/bts339

[5] Wood DE, Salzberg SL: Kraken: ultrafast metagenomic sequence classification using exact alignments.*Genome Biology* 2014, 15:R46.

[6] Langmead, B. (n.d.). How people build software · GitHub. *GitHub - BenLangmead/comp-genomics-class: Code and examples for JHU Computational Genomics class.* Retrieved from http://github.com/BenLangmead/comp-genomics-class

Distribution of Work (who did what):

Yu-chi (Kiki) Chang:

- Researched articles related to different ways to do exact matching and approximate matching
- Researched articles dealing with common contaminants
- Modified oneMismatch, twoMistach in matches.py
- Wrote Forward and Backward FMIndex class (not used in the final project)

- Wrote main.py
- Wrote abstract and revised prior work
- Structured and separated codes into different classes

Jeana Yee:

- Researched articles related to different ways to do exact matching and approximate matching
- Researched articles dealing with common contaminants
- Wrote Forward and Backward FMIndex class (not used in the final project)
- Wrote Prior Work, Methods and Software
- Modified oneMismatch, twoMismatch in matches.py
- Structured code into different classes

Mariya Kazachkova:

- Researched articles related to different ways to do exact matching and approximate matching
- Researched articles dealing with common contaminants
- Wrote code for algorithm to obtain correct indexes while using the forward-facing FM Index (discussed in methods and software; not used because idea changed)
- Wrote oneMismatch, twoMismatch, and naive in matches.py
- Wrote Introduction, Methods and Software, Conclusions

Min Geol Joo (Kevin):

- Researched articles related to different ways to do exact matching and approximate matching
- Researched articles dealing with common contaminants
- Modified oneMismatch, twoMismatch in matches.py
- Wrote main.py, analysis.py, analysis.m
- Processed genome and reads datasets

All members: met ~biweekly to discuss progress, setbacks, and to brainstorm algorithms.