Barcode Finder

Samuel Salas salas._@hotmail.com

April 10, 2023

1 Algorithm Description

The main idea used to solve this problem is Local Sensitive Hashing with Minhashing. The following explains the details of the implementation.

1.1 Main module

The logic behind the main module is very straightforward:

- Extract the sequences from the data
- Retrieve the anchor with the"get_anchor' function
- Use the retrieved anchor to get the raw barcodes from the sequences using"get_raw_barcodes function'
- Use retrieved raw barcodes to get the shortened list of barcodes.

1.2 Data Extraction

The extraction was done with pandas and the data was assumed to be uncorrupted. This means there is an iterative structure that follows the following order "read_id", "sequence", "+", "quality". Deviations from this structure will cause faulty output or crashes.
Module Time Complexity O(number of sequences * sequence length)

1.3 Anchor Finder module

This module strives to take advantage of the fact that, most of the time, a single anchor is used to extract sequences. The module counts the most common nucleotide per anchor position. It is expected that a random mutation would yield anomalous sequences. However, the most common nucleotide per position would be a part of the original anchor transcription. The barcode length and the unused nucleotides between the barcode and the anchor give the first anchor position. It is worth noting that the most detrimental mutations for this module are shifts in the anchor since it means a lot less indexes of the true anchor will

match with the anchor candidates. A justification for the validity of this strategy is explored in a later section.

This module only finds one anchor; however it could be improved to getting up to four anchors in time complexity O(n*m) where n is the number of sequences and m is the anchor length. This improvement is only possible if the probability of each anchor is not uniform. As the probability of each anchor appearing gets closer, this idea fails to retrieve a valid anchor.

If more than four anchors were needed the module logic would have to be changed and the same idea used to shorten the list of barcodes could be used to extract the anchors.

Another improvement possibility could be to take into account the quality of the sequence and discard the nucleotides that have poorer quality.

It is worth mentioning that if the anchor length is known for this data, this module could be bypassed, and the first barcode elements could be retrieved directly. This would only slightly decrease the raw barcodes' quality but increase the performance.
Module Time Complexity: O(anchor length * number of sequences)

1.4 Raw Barcode Finder module

To extract the raw barcodes from the anchor, the similarity between the sequence at the anchor positions and the true anchor are obtained using the Jaccard Similarity, which is defined as:

$$\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Each set is composed of shingles from the original sequence. Shingles are obtained by extracting all possible continuous substrings of size K from the original sequence at the anchor positions. Sets are compared, and a similarity threshold is used to discriminate between valid and non-valid anchor candidates. This way, anchor candidates that are similar to the real anchor are interpreted as valid anchors.

After an anchor is found to be valid, the next thing to know is to find whether there are any shifts on the anchor such that the barcode starts at"shift' positions from the 0th index. To do this, there must be an exact match between the real anchor's first E elements and the anchor candidate's first E elements. This is revised starting at the first position of the possible anchor start and then moved onward until an exact match is found. If the beginning of the anchor is corrupted, it is impossible to know if there was a shift that would make the raw barcode incomplete. There could be a mutation in the unused middle nucleotides, but there is no way to tell if those were mutated with additions or deletions since they don't have a predefined value.
Module Time Complexity: O(sequence length * number of sequences)

1.5 Barcode List Finder module

The module logic in this module is to:

- Get shingles of size S for all the barcodes extracted
- Get a minhash matrix using the shingles obtained
- Get Local Sensitivity Hashing (LSH) buckets from the minhash matrix
- Extract the true barcodes from the buckets that found similar band items in the minhash matrix and build a set with all of the ones that belong to a bucket of at least size Q

The minhash matrix is an H*N matrix where H is an arbitrary number of hashing functions and N is the number of barcodes. It is used to store the minimum hashing value among all shingles of each barcode for all the hashing functions used. This is useful because if two items in the matrix have the same value, it is extremely likely they share a barcode shingle.

The minhash matrix is divided into bands of size B, each with C columns (usual notation is R rows, but it is pragmatically slightly more uncomfortable). We use B hash functions to hash the C numbers obtained in the minhash function for each band. Our goal is to find if two band columns share the exact same C numbers. If they do, they are very likely similar. The hashed number obtained for a band column is called a bucket.

Once this last step is done, we can look into the buckets and search for the ones with multiple items. The more items a bucket has, the more likely it is that there is an underlying true barcode in the bucket.

Then, similar to the anchor module, we can take advantage of the fact that mutations occur at random places and extract the most frequent nucleotide per position to get to true underlying barcode. The barcodes obtained are then hashed to obtain a set of the unique true barcodes.

There are some fine tunning variables that are worth mentioning. First, the longer the shingles, the more similar the barcodes have to be mapped to the same hashing value. Second, the bigger B is, the smaller C is and the more similar two barcodes have to be groued to the same bucket.
Module Time Complexity: O(number of barcodes * number of hashing functions * barcode length)

2 Performance and Scalability

The total time complexity of this algorithm is:

$$O(number of sequences * number of hashing functions * barcode length)$$

This allows for great scalability, as the number of sequences is the most costly parameter. It could be worth translating this code to a faster language like C++ to get a roughly two orders of magnitude increase in performance.

3 Adaptability to New Data

This algorithm could adapt to new data if a unique anchor was used to obtain it. It would be easy to modify it to get up to four anchors for specific non-uniform probability distributions. It would be slightly harder to adapt if they are uniform or if there are more than four anchors. This last adaptation would decrease the performance as well.

4 Automatation

The program only needs the name of the file to yield a sound output. However, the fine-tuning needed to vary the quality of the results depends on the desired results as well. The next step for automation would be to include a parameter for the number of actual cells measured and run the algorithm with varying parameters to obtain the best result. If the number of barcodes is bigger than the number of cells tested, then it means there are similar barcodes being counted as unique. To fix this the minimum number of barcodes per bucket could be increased, B could be increased or the shingle size could be increased. If the number of barcodes is smaller than the number of cells, these parameters could be decreased. The type of error that produces a bigger or smaller number of barcodes could be used to assess which parameter to fine-tune exactly.

5 Justification

5.1 Theory

To assess the viability of the anchor finder we should take into account the possibility of mutations. If we assume the same rate of mutations in the anchor as in the barcodes, namely a hamming distance of 1, there should only be one nucleotide out of place. A more interesting case is if we assume a Levenshtine distance of 1. In this case, when we get shifts caused by deletions and additions we would still get half of the nucleotides in the right place. Substitutions produce a hamming distance of 1, which the algorithm can handle. If we assume a uniform probability of mutation, the most frequent nucleotide per position should be the one of the true anchors.

To assess the viability of the true barcode finder, we should take into account the limitations of the hashing used in the algorithm. A crash in two shingles that are different has a probability of:

$$\frac{1}{Hashspace}$$

This makes it very likely that two same hashing numbers have an equal shingle associated. The probability of having an exactly equal band for a barcode that is 80% similar is:

$$(0.8)^C$$

where C is the size of the band. For a band size of 5 we get a 0.382 chance of that happening. The probability that they are not equal in any of the bands and, therefore, two similar barcodes are found not to be similar equals:

$$(1 - 0.8^C)^B$$

Where B is the number of bands. If we get 20 bands, the probability that we won't find similarity is 0.00035. If the bands are 30% similar, the probability of finding a single identical band is 0.00243. The noise of these unwanted barcode similarities is reduced by setting a high threshold for barcodes in a bucket needed for consideration. The justification for this is the same as the one in the anchor finder.

Another way to reduce the probability of false positives and false negatives is by increasing the size of b. As the size of B increases, the likelihood of them decreases, but we compromise finding similar but not exact matches between sets.

5.2 Testing

To test the practical application of the program a set of X sequences are produced from a true anchor and a set of true barcodes. The sequences select a random barcode and attach it to the anchor using a desired number of unused nucleotides. The sequences are randomly mutated with a given mutation probability P. The possible mutations are "add" a nucleotide,"delete" one, or"substitute" one. There is also support for mutation of"any' type per mutation.

It is worth mentioning that deletion and addition are very similar since a deletion in a nucleotide requires addition of other nucleotides to preserve the sequence size. However, they are both preserved for future usability depending on scientists' specifications.

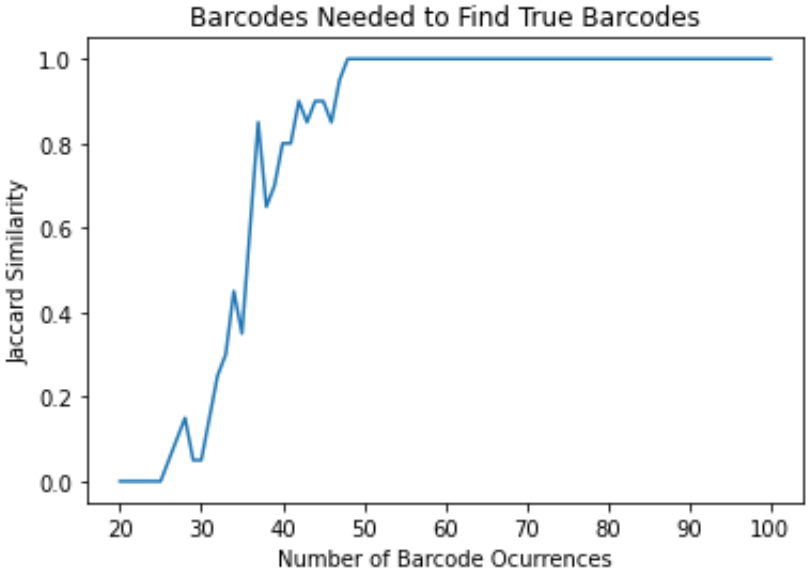
5.2.1 Anchor Finder

The anchor finder was tested using the randomly produced sequences for all mutation types and their combinations. The test was ran for a sequence size of 50,000 with an average of 2 mutations per anchor. In this test the program found the exact barcode every time for 20 iterations for all types of mutations. It is worth noting that if the parameters are changed to include more mutations and the test fails, the barcode is still very similar to the true anchor. This rough anchor is still useful for the problem because the valid anchors are meassured by similarity, not exact match. The caveat to this is that the anchor start could be corrupted and this would greatly impact the performance of the program.

5.3 Barcode Finder

This test assumes the true anchor was found. There is also an average of two mutations per barcode in the dataset used. An important question for this algorithm is "how many repetitions of the possibly mutated barcodes are needed to obtain the true barcode?". With the parameters that are present by default a test was run to get an idea of this number.

To test the threshold of the minimum number of barcode repetitions to obtain the true barcode, an ideal case of one barcode was made. A varying amount of sequences was produced and the similarity between the obtained barcode was be assesed against the true barcode. The following graph expresses the results. Each case was run for 20 iterations. A minimum of 20 barcodes per bucket is required, this less that 20 sequences is not worth testing.



The barcode finder was also tested with 50,000 sequences produced from 500 barcodes with a mutation probability of 0.5 which comes out to the rougly 2 mutations per barcode. All mutation types and combinations of them were evaluated. The test ran successfully, finding all true barcodes in every case.

6 Last Remarks

The performance of this program can be highly variable depending on the parameters. This variability can be greatly reduced by knowing the nature of the data produced. Close communication with scientists could help fine tune the parameters to get good performance. Otherwise, if the data is highly variable, the parameters would have to be automatically tuned, and the program should be run many times to produce the best-performing result, measured by its closeness to the desired number of barcodes.

To get a fine-tuned version of this program, another possibility is to use real data. Using even the desired number of barcodes expected with many datasets could give a good estimate of the true performance of this algorithm. Because of the lack of this data, the parameters used in this implementation can still be greatly improved!