**Details of the Implementation of our Method**

Implementation of our method is an automated pipeline with no human intervention from start to end. We designed it in parts so after finishing, one could restart from any middle step. However here we focus on some key points that we think either algorithmically or practically will be useful to anyone who wishes to rewrite and improve our method.

We will go over implementation on the same dataset that we used in the comparison study; 50 HCV 1a genomes as target genomes and 50 HCV 1b genomes as non-target genomes, however with tighter Tm ranges. Also, here we use more than one seed genomes to show generation of consensus genome.

After pre-filtering, we want to construct a consensus genome. It is an iterative process that with Mummer4, we align the resulting common oligonucleotides with the new genome in a pairwise manner. We find common regions of length >=15bp with the mummer command: # mummer -maxmatch -l 15 -n save location+id ref\_file qry\_file # save command will generate files as id.aux id.isa id.kmer id.lcp id.sa, after the first run, instead of ‘save’ we use ‘load’ with the same id.



Figure 1: Common regions of first two genomes

After some scripting, we extract all common subregions and locations. If there are more than two seed genomes, later it is important not to query all strings in batch mode because we noticed that Mummer4 has a bug that may miss already present strings, but this bug is present only in the batch mode. Querying oligonucleotides is present in many stages of our method, therefore in every phase of our implementation, we query all distinct strings separately, then combine them.

As we mentioned in main article, this common region forming step is optional and if one seed genome is used, that whole genome becomes the consensus string that oligonucleotides will be extracted from. Using only one seed genome must be default process for highly variable viruses because constructing consensus genome may and will lose very important information.

Then we extract oligonucleotides from the common region. It is simply a two-way string scan. We report all primers and probes with their orientation information along with the Tm and locations. Then we group similar strings together and give them a group id. We used a simple logic, when we extract a string whose starting point is not between start and end points of previous key string (first string of a group), that string becomes a key string. There may be better ideas than this for forming groups. Because we used tight Tm ranges in this demo study, Figure 2 shows a small number of strings in some groups, however number of oligonucleotides in a group can be high up to hundreds. Grouping is done separately for primers and probes, also separately for orientations.

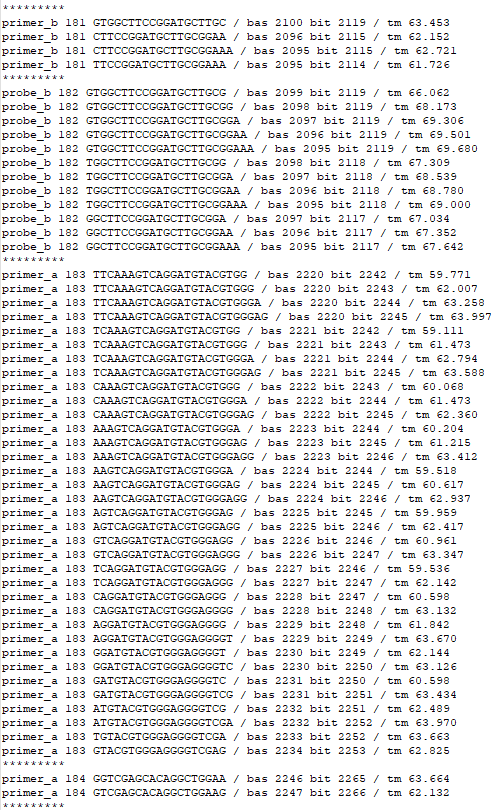


Figure 2: Extracted oligonucleotides

Then very carefully with start and end locations, we construct possible amplicons considering different constraints such as maximum allowed primer Tm difference, probe-primer Tm difference, amplicon length and some dimer constraints.

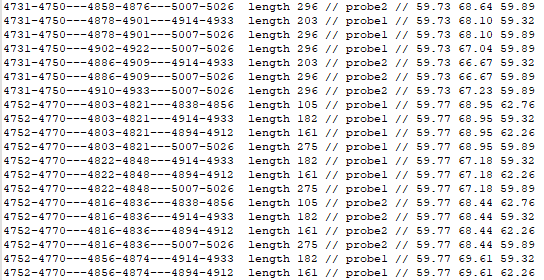


Figure 3: Constructed amplicons on seed genome

In the next important step, we choose a fixed number of oligonucleotides from every group and we look for amplicons that are formed by these oligonucleotides in both target and non-target genomes. Here is one important detail, we do not pick amplicons from all possible amplicons and then query those oligonucleotides in other genomes; but we pick oligonucleotides from previously formed oligonucleotide groups. The reason is the following: as seen in Figure 2 some oligonucleotide groups may have many more oligonucleotides. As an extreme example, one primer\_a (orientation a) group and one primer\_b group in close proximity allowed by amplicon length both may have 1 single string and those two groups would generate 1 amplicon. However other two groups having each 100 strings will generate 10000 amplicons. So, sampling randomly from amplicons may generate a very non-uniform search space. We could have sampled a fixed number of amplicons from combinations of groups and it would be needlessly complicated and it already is close to what we did. However, there may be other ideas. Now we have all oligonucleotides to be queried on all other genomes, target or non-target. We now query every oligonucleotide separately to every single target and non-target genomes. This process is done parallel, every CPU studies one genome and switch to one other unprocessed genome when finished.

Now instead of the mummer command, we query with nucmer command that allows mismatches. We use the command: # nucmer —maxmatch -l minimum\_length\_of\_anchor -c minimum\_length\_of\_alignment –nooptimize –save location+id ref\_file qry\_file #. This part is the heart of the algorithm and the most important reason why our method is so successful.

All variables of suffix array queries are explained in detail in experimental results section, however there is one detail we would like to add. We parameterized minimum length of alignment based only on length of oligonucleotide to be queried and his works very well for general temperatures used in PCR studies. However, if one wants to generate smaller length oligonucleotides with low Tm, we suggest to use a baseline minimum value such as maximum of (a constant value, length of oligonucleotide-k).

Here we first generate dummy out files for every query string and then concatenate to one file for each genome.

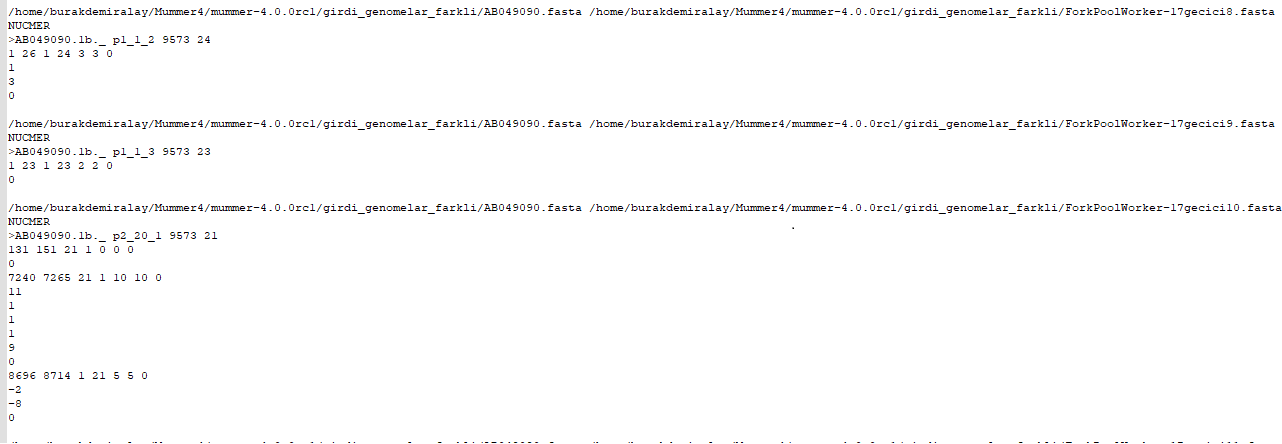


Figure 4: Suffix array result of queries

Explanation of output file shown in Figure 4 is as follows; 7240 7265 21 1 10 10 0 #

Each of these headers is followed by a string of signed digits, one per line, with the final line before the next header equaling 0 (zero). Each digit represents the distance to the next insertion in the reference (positive int) or deletion in the reference (negative int), as measured in DNA bases.

So we now can extract hit locations, sometimes multiple hit locations for every oligonucleotide for every genome.

After extracting all hit locations for every oligonucleotide, we can look at their interaction thermodynamically, most important factor. We generate a file for every genome, target or non-target, Tm of every oligonucleotide and interaction location and orientation. This process is also fully parallel and every CPU holds one genome and switches to next after finishing.

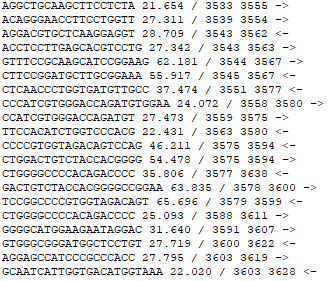


Figure 5: Thermodynamic interaction results of oligonucleotides with a random input genome

One improvement that can be done is that, we can combine suffix array query and Tm analysis, and instead of querying every single oligonucleotide to a single target genome in a CPU, we can instead query one oligonucleotide against every genome. In that case, as soon as the hit count violates the required limits, the program would proceed to another oligonucleotide. Since the purpose of subtype analysis is to find a small region that is vastly different and a substantial portion of genome regions are similar, this would reduce running times significantly.

Now in the final step, according to desired true and false positive rates, we output a final report file that contains every single amplicon in every genome if there is an amplification *in silico*. This process is also parallelized on every CPU and there are even multiple threads on every CPU because this process is very reading and writing intensive.

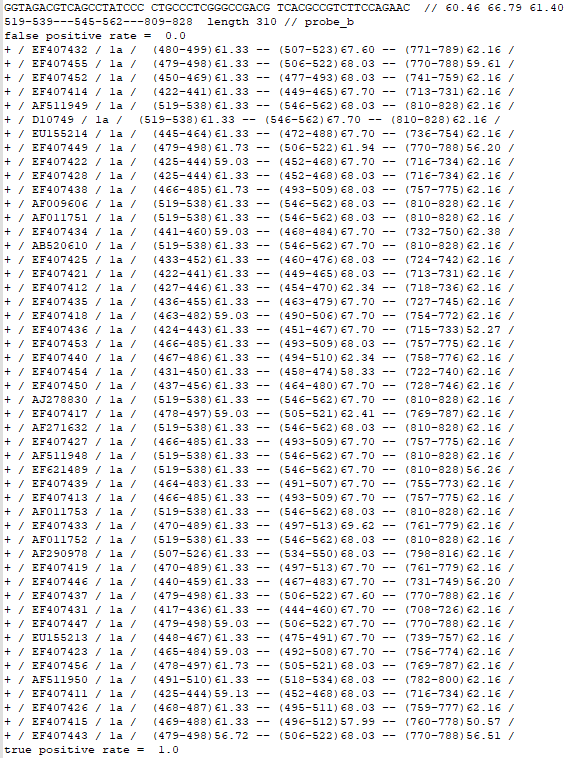


Figure 6: End Result