AlignFix

Group Members: Siddharth Kaipa, Manish Sampath, Jason Chiu

Introduction: Alignfix is an aligner that uses a seed and extend heuristic to align short query sequences to a genome. The goal of this tool is to align a sequence to a reference database and return an optimal alignment decided by an internal scoring function. The tool can be applied to a variety of biological problems. One of the primary applications of a tool like this would be to align short, high-fidelity sequencing reads to a large database. Another function of the tool would be to search for functional and evolutionary relationships of a sequence. For example, one could query a string from the mouse genome using a human genome as the reference database to look for orthologs. The functionality of this is limited by the fact that our tool requires at least a 15-mer match between the query and the reference.

Methods: The implementation of AlignFix can be described by a seeding step and an extending step.

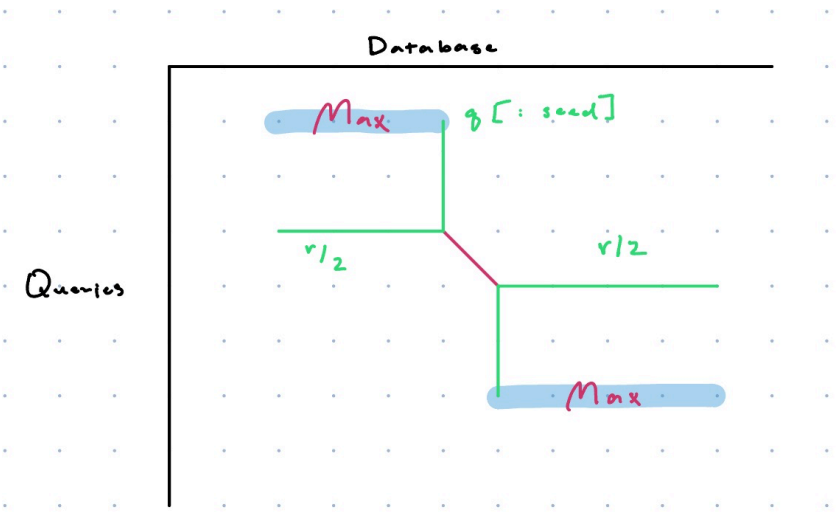
The seeding step is attempting to find some 15-mer match between the query and the database. This is equivalent to a fast pattern matching problem. To solve this problem, we have decided to pursue an approach where we preprocess the database into suffix arrays. To generate suffix arrays, we append “$” to the end of the reference genome or database in general. For example, if the database was the word “hello”, the preprocessing step would change it to “hello$”. The next step was to generate all possible suffixes of the input. The suffixes would be [hello$,ello$,llo$,lo$,o$,$]. Then, we sort the array lexicographically and end with the following [$,ello$,hello$,llo$,lo$,o$]. Finally, identify the index in which these suffixes show up: [5,1,0,2,3,4]. [2]

After preprocessing the database into suffix arrays, we use binary search to find seeds in the genome. We slide a 15-mer window across the query until we find a set of seeds and break. The efficiency of this algorithm rests in binary search. Binary search allows us to divide the search interval in half by narrowing the search of the element to the upper or lower half until a match is found, resulting in an O(log(n)) instead of O(n) excluding the preprocessing step. We do binary search twice. The first time to find the first occurrence of our 15-mer match in the suffix array and a second time to find the last occurrence of our 15-mer match in the suffix array.

After completing this step, we have found the indices in the genome corresponding to our 15-mer match and can continue to an extending step.

With the set of seeds, we will return an alignment that maximizes the output of our scoring function. Briefly, the scoring function we use is an affine gap penalty scheme, where matches are set to +2, mismatches are -3, gap open penalties are -5, and gap extension penalties -2.

The affine gap penalty allows us to extend a gap in the alignment without penalizing too harshly as deletions might encompass a certain span of the query. The solution we discuss borrows ideas from the Smith-Waterman algorithm with modifications. The query and the region of the database are put into 3 “layers” of 2D matrices; the upper and lower layers contain extension penalties while the middle layer contains the match reward and mismatch penalty. Hopping from the middle to any other layer incurs a gap opening penalty. Once the matrices are scored, we implemented a simple backtracking method to generate the alignment based on where the max scores came from.



Having described how the alignment works, we will now explain how we stitched together these two algorithms to create a cohesive aligner. Refer to the image above. The red line indicates the matches given by our seed. We define two alignment subproblems: bottom alignment (bottom right corner) and top alignment (top left corner). Note that we include r / 2 length of the database to the alignment subproblems. We decided that the best value for r in the program was 2 x the length of the query. The idea is that we want to make sure that enough of the database is given

to the alignment subproblem without wasting memory and increasing the runtime. As for the query, we hand over the rest of the query to the problem depending on which side of the seed the alignment problem rests. For bottom alignment, we proceed with the dynamic programming strategy described above. However, for top alignment, we first have to reverse the database string and the query string that we hand to the alignment problem, proceed with the dynamic programming strategy, and then reverse the alignment strings. Finally, we sum up the scores of the top and bottom alignments to produce an overall score. We also concatenate all alignments and the seeds. Lastly, we calculate the start and end position of the alignment in the genome.

We designed three benchmarking experiments. First, we simulated a number of genomes with various lengths from 0 bp to 160000 bp and 50 reads of length 100. This allows us to analyze the runtime and how it scales with the genome. Second, we simulated a single genome of size 20000 and created a set of reads that varied from 20 reads to 1600 reads. This allows us to analyze the runtime and how it scales with the number of reads. Finally, we simulated a genome of size 20000 and a set of 50 reads with varying levels of percent error from 0 percent error to 29 percent error. This allowed us to analyze the percent of aligned queries as the fidelity of the reads decreased.

To make the package user-friendly, we implemented command line arguments that can run our program in the terminal. In our main file, we utilized the argparse method to take in arguments via the command line. There are three main arguments, denoted by –genome, –query, and –output. The structure in which the command should be typed is “python alignfix --genome genome.fasta --query queries.fasta --output output.txt“. To allow the user to type in “alignfix” instead of “[path to root]/alignfix.py,” I followed a post from Stack Overflow [4]. This way, the user can easily input any genome they want along with what queries they want to search for and output the alignment results in a text file.

To periodically test our methods, we went to the European Nucleotide Archive [5] to get our reads and the National Library of Medicine [6] to get our genome. From there, we wrote a method which would take reads of a specified length k from the genome and randomly mutate it. This way, we could check how the tool functions if there is no exact match.

In our package, we had to install Numpy v1.26.4 and pyfaidx v0.8.1.1. Numpy was imported to create and work with arrays/matrices. We used them to create suffix arrays and scoring alignment matrices. Pyfaidx was used to read and interpret fasta file inputs.

Results:

We created multiple benchmarking experiments, each revolving around one of three different parameters (number of reads, genome length, and chancer of read error). We observed how they affect the runtime and quality of our results using a graph. When comparing the number of reads to runtime, the graph shows a linear upward trend, indicating that the runtime is O(n). When comparing the genome length to runtime, the graph shows a O(nlog(n)) runtime. This makes sense because there is a sorting step involved with the preprocessing step of the genome. When comparing the chance of error to the percent of aligned queries, the graph shows a linear downward trend, indicating that the more errors a read has, the fewer aligned queries there are.

```
>ERR10000000.587795
CCTTGAAGGTTCTGTTAGAGTGGTAACAACTTTTGATTCTGAGTACTGTAGGCACGGCACTTGTGAAAGATCAGAAGCTGGTGTTTGTGTATCTACTAGTGGTAGATGGGTACTTAACAA
CCTTGAAGGTTCTGTTAGAGTGGTAACAACTTTTTATTCTGAGTACTGTAGGCACGGCACTTGTGAAATATCAGAATCTGGTGTTTGTGTATCTACTAGTTGTAGATGGGTAGTTAACAC
Score: 170
Start position in database sequence: 30194.0
End position in database sequence: 30414.0
>ERR10000000.587796
TGAAAACATGACACCCCGTGACCTTGGTGCTTGTATTGACTGTAGTGCGCGTCATATTAATGCGCAGGTAGCAAAAAGTCACAACATTGCTTTGATATGGAACGTTAAAGATTTCATGTC
TGAAAACATGACACCCCGTGACCTTGGTGCTTGTATTGACTGTAGTGCGCGTCATATTAATGCGCAGGTAGCAAAAAGTCACAACATTGCTTTGATATGGAACGTTAAAGATTTCATGTC
Score: 405
Start position in database sequence: 29330.0
End position in database sequence: 29550.0
```

The above is a snippet of the output when we ran our tool on the covid genome mentioned in the above section. You can find the links to these datasets in the README. In general, our output has the alignment, the score we calculate, and start and end coordinates in the genome. The results are meant to look like BLAST. [3]We have a more thorough explanation in our README. Notice that our program does a good job handling mismatches as seen in the first entry in the snippet. Upon further inspection of the output, one will find that the program will also handle indels extremely well.

We compared our tool against BWA mem using a covid dataset. The results were as followed:

Categories	AlignFix	BWA MEM
Time To Align	152.93 seconds	0.289 seconds
Percent of Queries Aligned	83%	99.93%

BWA is significantly faster than our tool. It also aligned more queries than AlignFix. However, since our original goal was to create an alignment tool that would align at least 80% of queries, we succeeded in our goal.

Discussion: One of the hardest parts of this project was figuring out how the affine gap penalties should be coded, as we needed to take into account 3 matrices, not just one. Another challenge was figuring out how to get the command line arguments such as pip install to work. To fix this, we looked at stackoverflow for advice [4]. Once those were fixed, the tool worked as intended. For future directions, it would be nice to see this tool work with other file formats other than fasta. We would also like to test different lengths for the seed and analyze how it impacts sensitivity.

Code: https://github.com/msampath25/alignfix

References:

[1]https://www.sevenbridges.com/short-read-alignment-seeding/#:~:text=Alignment%3A%20a%20quick%20review&text=Many%20modern%20alignment%20algorithms%20rely,so%20seeding%20is%20very%20fast.

[2]https://usaco.guide/adv/suffix-array?lang=cpp

[3]https://blast.ncbi.nlm.nih.gov/Blast.cgi

[4]https://stackoverflow.com/questions/56534678/how-to-create-a-cli-in-python-that-can-be-installed-with-pip/66010978#66010978

[5]https://www.ebi.ac.uk/ena/browser/view/PRJEB37886

[6]https://www.ncbi.nlm.nih.gov/nuccore/NC_045512.2