Willard Ford
Group 26

Project Report: wf-align

Introduction:

Current tools to align reads to reference genomes have near optimal efficiency. Tools such as BWA-MEM and Bowtie2 both run in near-linear time complexity and linear space complexity to align DNA reads to a reference genome. STAR alignment achieves similar results for mapping RNA reads. However, the advent of the draft human pangenome requires a new set of tools and algorithms to align reads to a new graph-based reference genome structure. Good science should seek to incorporate more representative tools such as the human pangenome because they will lead to more representative discoveries and eventual healthcare benefits for huge portions of the population. Therefore, the discovery of new algorithms to align reads to the human pangenome is extremely important.

I did not design that tool, but I want to eventually help in its creation. To assist in my own learning, I built a read alignment tool, *wf-align*, to exactly align single-ended reads to a one-dimensional reference genome. wf-align outputs a sam file containing the first exact match for each read in the input fastq file. This tool can effectively align reads from fastq files to a reference fasta file if the matches are exact in O(length of reference * length of read) time complexity and O(length of reference genome) space complexity. Because my tool only aligns exact reads, its practical application is limited, with varying effectiveness based on the quality and lengths of the reads. Due to the time complexity of my tool, it is efficient to align genomes in the length of viruses and many shorter animals, but takes too long to align reads to the human genome.

Methods:

To achieve O(length of reference * length of read) time complexity and O(length of reference genome) space complexity wf-align uses a unique algorithm based off of BWA-backtrack with a major difference noted below which accounts for the superlinear time complexity. All code is written in python and references cited in the documentation if necessary (in radix sort and suffix array methods).

wf-align generates a number of auxiliary structures for each chromosome from the input fasta file. After reading in a reference chromosome string, wf-align first generates a suffix array using a method called prefix doubling (effectively described in this video:

https://www.youtube.com/watch?v=_TUeAdu-U_k). It maps a reference genome into an alphabet of numbers maintaining the same relative comparative order. Prefix doubling makes sets of 2 consecutive elements and sorts each suffix by their first 2 characters. Then by repeating the process n times using the 0-th and 2^n-th character as our sets, prefix doubling repeats this process log_2(n) times to build a suffix array. By implementing radix sort this step takes only O(n*log(n)) time complexity and O(n) space complexity.

From the suffix array wf-align generates a burrows-wheeler transformation vector in O(n) time and space complexity. Then from the burrows-wheeler transformation vector it regenerates the first column of the burrows-wheeler transformation matrix by directly sorting the burrows-wheeler transformation vector in linear time and space complexity using radix sort again. By doing so with an auxiliary data structure it can store the original and sorted position to generate a last-to-first vector that will be used during alignment. Finally, wf-align records the index of the first occurrence of each unique character in the first column of the burrows-wheeler matrix in a vector called C. This is done in linear time and constant space because the alphabet is constant. Subsequently, wf-align discards the first column of the burrows-wheeler transformation vector.

The burrows-wheeler transformation matrix has an inherent property that for each row, the last character in that row is followed by the first character in that same row. This is because the burrows-wheeler matrix is the ordered cyclizations of the given string. Then because a stable sort is used to generate this data, the n-th occurrence of a given character in the first column, corresponds to the n-th occurrence of that same character in the last column of the burrows-wheeler transformation matrix. By moving from a character in the last column to the string's next character in the first column the burrows-wheeler matrix can rebuild 2 characters of the string. Stitching these 2 character sequences together with the last-to-first vector, the entire string can be constructed. Rebuilding the input this way is called backtracking.

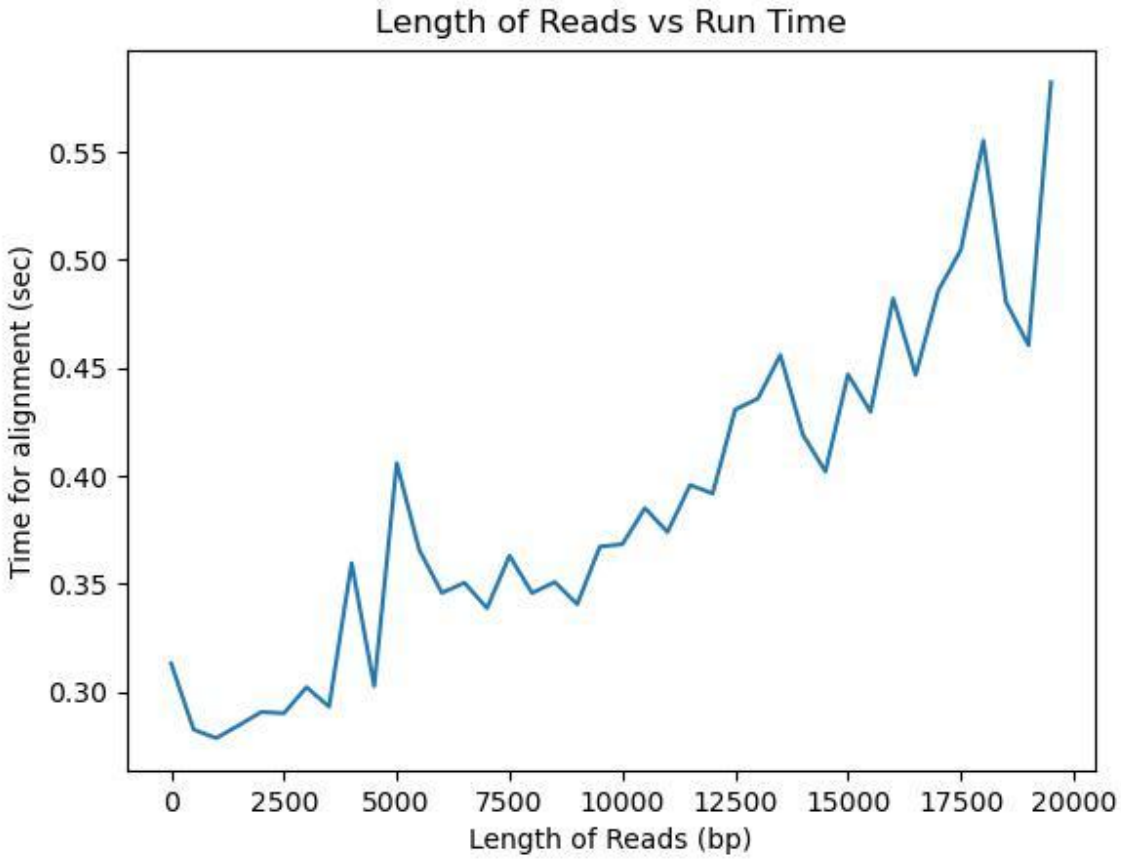
To perform the actual alignment, wf-align starts with the last character of the query/read string and places a min and max index for the corresponding character in the burrows-wheeler vector using the C vector in constant time. This range corresponds to the possible indices of exact matches of the query string. Then to find the next range of valid characters, wf-align checks each character in the range of the burrows-wheeler vector and stores the min index and max index of the next matching character. This is where wf-align differs from BWA-backtrack and scales with the size of the reference genome. Finally, by using the last-to-first vector to find the new min and max range of possible exact matches wf-align can repeat the process and find the query string. Once it discovers that there is either no exact match wf-align skips to the next

read. If it discovers a range of matches, we can convert burrows-wheeler indices in the vector back to the indices in the original query string using the suffix array. We repeat this process for each chromosome or until a match is found.

To prove these results I randomly generated datasets that vary the length of the reference genome and the number of reads to isolate changes in either group and record how long our algorithm takes to search. I used short read lengths to make sure that there is an exact match available or else our algorithm will terminate too quickly to see a difference across the varying reference sizes and number of queries.

Results:

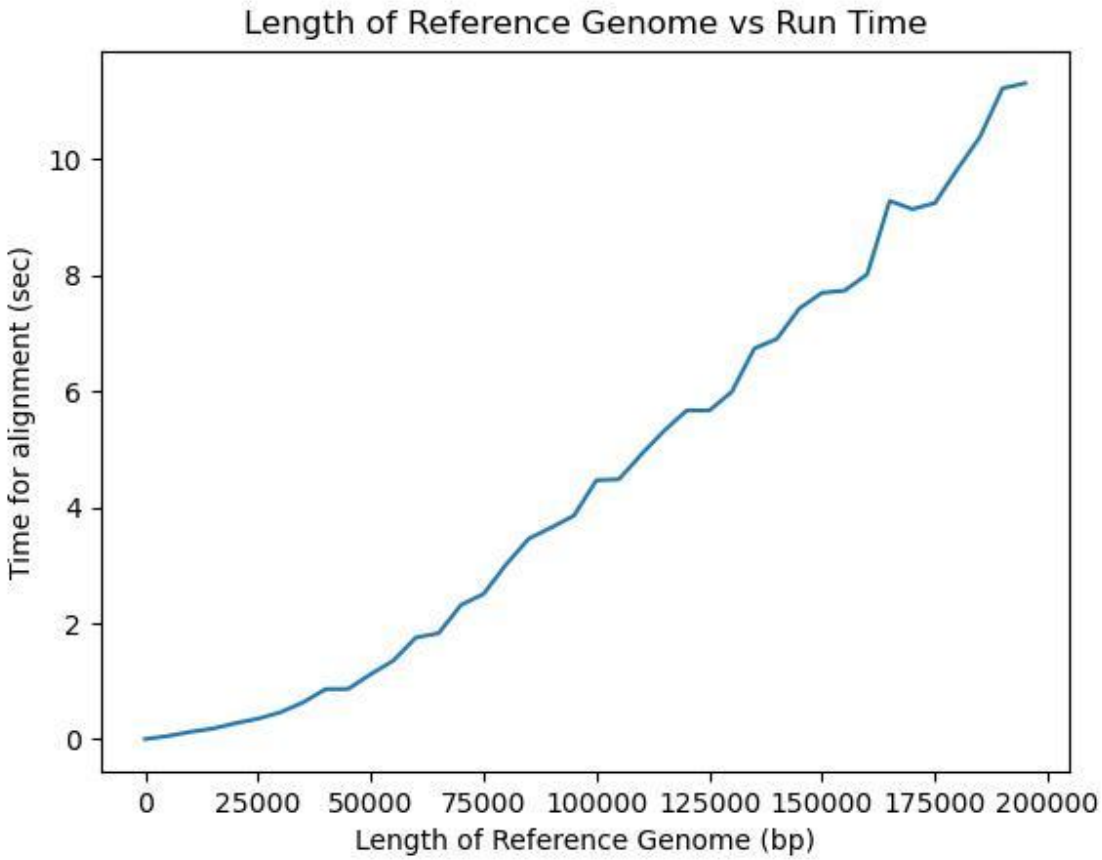
When using a randomly generated reference genome and varying the number of reads we can see that the run time scales linearly with the total length of reads shown below.



We see a decent amount of noise because the size of the reference genome in this example was relatively small at only 20,000 base pairs.

Similarly, we can see that when varying the length of the reference genome we also see a close to linear increase in the amount of time necessary to search for exact matches when

using randomly generated data. This isn't exactly linear because generating auxiliary data structures also depends on the length of the reference genome in O(n*log(n)). The example below shows runs with 200 reads, each of length 10.



We can run wf-align on the SARS COV2 reference genome using the command: "wf-align sequence-4.fasta ERR10000004.fastq -o output.sam -m metric.txt". The SARS COV2 reference genome is 29,903 base pairs long and the reads have a total length of 1,157,474 base pairs. wf-align aligns 18.9 % of reads map back to the genome in 4.65 seconds, this low percentage can be attributed to only allowing exact reads to align.

We can run BWA-MEM on the same data using the following two commands: "bwa index sequence-4.fasta" followed by "bwa mem sequence-4.fasta ERR10000004.fastq". The first step builds the auxiliary data structures in 0.044 seconds and the second step performs the alignment in 1.920 seconds, totaling for 1.964 seconds. This means BWA-MEM runs about 2.4 times faster than wf-align on this relatively small example, and we would expect this gap to grow when using larger reference genomes. BWA-MEM only failed to map 3 reads resulting in

5361/5364 reads being mapped, a much higher percentage than wf-align which can be attributed to allowing the mapping of inexact reads.

Discussion:

This project was a great learning experience for myself. The biggest struggle I encountered was trying to balance which algorithms would be best to implement for space and time efficiency versus which ones I could actually learn and implement in a reasonable amount of time. For example, linear time burrows-wheeler construction algorithms exist through methods such as induced sorting, but without the requisite background knowledge I had a hard time actually implementing these methods. I think the level of optimization I settled on was a good compromise but given more time I would like to try implementing wavelet trees rank function for linear search time.

Ultimately, the reason I was interested in this project is because I want to help build the new algorithms and data structures that will need to be invented for alignment to a non-linear human pangenome. Starting from almost no knowledge, however, I decided to try and learn how alignment is currently done. But eventual future directions should be the discovery of new algorithms.

Finally, this code was all written in python and bash, though when optimization is most important the tool should be written in C or C++. The reason for this is simply because I'm not familiar with C or C++. Given some time, my next tool should fix this problem.

Code Availability:

https://github.com/WillardFord/wf-align-CSE185

References and Research:

Ben Langmead Burrows-Wheeler Indexing Lectures:

https://www.youtube.com/playlist?list=PL2mpR0RYFQsADmYpW2YWBrXJZ_6EL_3nu

Bowtie2

https://bowtie-bio.sourceforge.net/bowtie2/manual.shtml#the-bowtie2-aligner

BWA MEM:

https://bio-bwa.sourceforge.net/bwa.shtml

CMU Design & Analysis of Algorithms Lecture Notes:

https://www.cs.cmu.edu/~15451-f18/lectures/lec26-suffarray.pdf

https://www.cs.cmu.edu/~15451-f18/lectures/lec25-bwt.pdf

Niema Moshiri CSE 100R Lectures:

https://www.youtube.com/watch?v=Lc-ACiJIrnM

https://www.youtube.com/watch?v=IzMxbboPcqQ

SARS COV 2 FASTQ Reads:

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

SARS COV 2 Genome Analysis:

https://doi.org/10.1016/j.genrep.2020.100682

SARS COV 2 Reference Genome:

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

STAR Aligner:

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3530905/