Investigations into the Compression and Analysis of Genomic Data

Michael Gee, Edward Garemo, Antara Pattar

Abstract

In this paper we investigate the compression of genomic data, and the possibility of applying algorithms directly thereupon. We look into the feasibility of compressing arbitrary genomes, concluding that there is unlikely to be much room for improvement beyond using a 2 bit encoding. We then present a successful algorithm for compressing datasets composed of many genomes/genomic sequences, which at least on certain datasets seems to outperform the state of the art. Finally, we investigate the possibility of doing analysis and running algorithms directly on compressed data, and present a particular instance of us doing so.

Introduction

Over the past few decades the cost of sequencing DNA has dropped exponentially, which has resulted in a massive explosion in the amount of genomic data available. Illumina alone has sequenced more than 500PB worth of data (Illumina 2020), which, based on a back of the napkin calculation, we estimate would cost upwards of $100 million to store in raw form. This makes the potential of reducing genomic file sizes very attractive.

Data of this size also poses problems for analysis, since few computers have RAM memories large enough to store, for example, a hundred human genomes, creating a barrier for real time desktop analysis. Applying algorithms directly to compressed data, rather than to data in its raw form, opens up the possibility of analysing larger datasets more quickly, which in turn could power more powerful analyses and discoveries.

This paper explores the efficacy of various compression techniques on both individual genomes and sets of similar genomes, and explores the possibility of performing analysis on compressed data.

The first section presents attempts to compress individual genomes by finding unifying features in genomes from different domains. This is achieved through the attempted construction of a ‘universal reference’, a term which will be explained later.

The second section of the paper investigates the compression of large collections of genomes from the same species, and presents an algorithm which achieves compression rates upwards of 500x.

The third section considers applying algorithms directly to compressed data and investigates the difficulties associated with doing so. Extended work is presented on a compression approach based on the Burrows-Wheeler Transform, and an algorithm that can be applied to that compressed data, in order to demonstrate these ideas. Challenges and observations associated with applying algorithms to compressed data are discussed.

Code

The code used in this project is hosted at https://github.com/EdwardG5/Project85.

Section 1: The Compression of Arbitrary Genomic Data

Introduction:

The NCBI RefSeq database contains reference genomes for over 100,000 distinctly-named organisms (RefSeq 2021) and the NCBI GenBank database contains sequences for over 250,000 distinctly-named organisms (GenBank 2021). Being able to compress these individual records would greatly increase the speed at which researchers could access data and reduce the cost of storing the central repository of data.

While great success has been found compressing groups of similar genomes, current state-of-the-art techniques for compressing individual genomes using statistical methods (Sulistyawan, Arifin, and Fatoni 2020) have not made significant improvements over the compression rate achievable by a bit-wise representation of the genome, instead of ASCII characters (Boufounos, El-difrawy, and Ehrlich 2002).

Currently multiple alignments are able to achieve incredible $\approx$ 400-fold compression by utilizing **universal reference compression**. Universal reference compression is the current state-of-the-art solution for compressing genomes of similar construction (Huang et al. 2017). This is a proven method for applications such as multiple alignment or species compression.

However, there does not exist a reference for reference-based compression that is agnostic to species or alignment, nor much literature explaining attempts at such a method.

In this section, we share attempts at curating a universal reference for bacterial genomes and ultimately discuss why these approaches are unsuccessful compressing individual genomes. Bacterial genomes were utilized due to their shorter length relative to other genomes for faster analysis.

Universal Reference Compression

What is Reference-Based Compression?

Referenced-Based Compression is the utilization of a **reference** set of data in order to compress other, **target** data. Both the compressor and decompressor have access to the reference data; so operations like the example in Figure 1 are possible.

Assumptions

In this section on reference genome compression, we assume that FASTA files contain only the characters `A, C, T, G, N,` and `-`. This greatly reduces the complexity of the problem for the purpose of our study, yet is easily scaled to the full ensemble of FASTA nomenclature.

Reference: AAATCG
Target: AAATNATCG
Compressed: [(0,4),N,(2,4)]

Where each tuple (a, b) in the compressed list contains:
a: The starting index of the target in the reference
b: The length of the copied substring
Elements that are not present in the reference or not long enough to be effective to compress are untouched.

FIGURE 1. *Example of Reference Compression*

Dictionary-Based Universal Reference

A dictionary-based universal reference is a reference used by a compressor to compress data with a **dictionary data structure**.

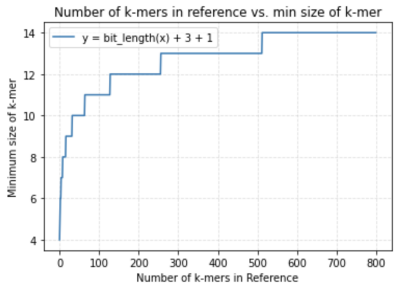
In the dictionary-based universal reference approach, we construct a dictionary of N k-mers, for integers k and N. When a k-mer that exists in the dictionary is encountered in the genome, we insert a 3-bit reference indicator followed by the bitwise index of the k-mer in the dictionary. References are matched greedily as the target genome is traversed for fast, albeit sub-optimal, compression.

The benefit of a dictionary-based reference compared to a string reference is that only one integer, the index of the string in our dictionary is required. A string-based approach requires encoding both the starting index and length of the substring, which are both typically larger numbers that the dictionary index.

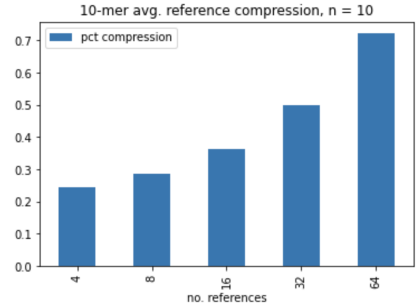
Studying the *10bacteria* dataset (Annex A), it was determined that >5% of the genomic strings could be represented by just 20 6-mers. Hence, a dictionary-based reference approach was formulated.

Since only a small subset of ASCII characters are used in genomic data, simply using fewer bits to represent each character can achieve remarkable compression. Three bits is enough to represent eight different characters (enough for the four nucleotide bases, plus extra for some of the more common FASTA characters); using a three-bit representation alone can compress files to 37.5% of their original size. For this reason, we utilize a 3-bit genome encoding. Hence, the analysis presented in this section is benchmarked against a 3-bit encoded genome, which is already 37.5% of the original size.

So, when constructing our dictionary, we plot in Figure 2 possible values of N and k in order to yield effective compression. We derive the minimum size of k-mer to go in a reference dictionary that holds N k-mers by considering the number of bits required to represent indices in the dictionary, the 3 bits required to flag the utilization of the reference to the compression (using 3-bit characters), and 1 bit in order for compression to be effective (not the same number of bits as k-mer would have without compression).



Number of k-mers (N) able to exist in reference for each k.



Percent compression for varying 10-mer dictionary sizes.

FIGURE 2. *Dictionary-based Reference Analysis*

Testing this with $k = 10$, we test the performance (percent compression) at varying numbers of 10-mers in the dictionary N. Figure 3 plots the average performance, with 0.72% compression with a dictionary reference containing 64 10-mers, across the *10bacteria* dataset (Annex A), containing 10 bacterial genomes. The k-mers utilized in this dictionary were sampled from a distinct random sample of 100 bacterial genomes, the *100bacteria* dataset (Annex A).

Suggested Improvements to Reference Dictionary Approach

When benchmarked against a 3-bit genome encoding, the dictionary-based compression approach achieved a 0.7% compression ratio using a reference with a total length of only 640 characters. The efficacy of this approach could be improved by sampling larger k-mers. The universe of k-mers exponentially increases as k is increased. Hence, sampling and aggregateing the frequency of k-mers quickly across a set of genomes, for large k, requires additional memory resources.

Currently, substrings are assigned greedily as the target genome is traversed. However, there does exist some optimal encoding of the genome which could be found utilizing a dynamic programming solution, with the additional computational resources.

Non-Dictionary, String-Based Universal Reference

The dictionary-based reference provided a compression that, if upscaled (for k and N) with the necessary computational power, displayed promising results for universal compression. However, it lacks the ability to capitalize on compression of contiguous k-mers. So, we propose a string-based universal reference that does not possess this weakness.

In this string-based universal reference compression approach, we construct a reference string *S*. We scan some genome *G* linearly, from start to finish. At each character in *G* that has not been encoded by a previous iteration of the algorithm, we find the longest substring of S that matches a substring of G starting at that character. We encode this compression with 3 bits indicating utilization of the reference followed by the start index of the substring in S and length of the substring using a standardized number of bits (generally 8). Substrings are only encoded if they contribute favorably to compression.

We present an example in Figure 3, assuming the encoded result contains fewer bits than the 3-bit encoding of the genome.

Generally, reference based compression is utilized for multiple alignments or genomes from the same species (Kozanitis et al. 2011, Deorowicz, Danek, and Niemiec 2015), so there does not exist literature or tools to run string referenced-based compression efficiently.

We propose the *Algorithm 1*:

Algorithm 1: Production-ready Greedy String Reference-Based Compression

```
Result: String-Reference Compressed Genome
Construct a trie of all substrings in S;
for character in G do
    Greedily match largest substrings with substring in S;
    if Compression has smaller bitwise representation than 3-bit encoding
     then
        Encode with 3-bit reference indicator, start of substring in S, length of
          substring;
    end
    else
        Encode substring with standard 3-bit representation;
    end
end
```

This algorithm performs quadratic preprocessing in Step 1, allowing for linear lookups at runtime. This allows for fast, production-ready applications by doing quadratic preprocessing before runtime.

Reference: `AAAAAACGT`
Genome: `AAAACG`
Encoded: `(Ref indicator)(dec2bit(2))(dec2bit(6))`

FIGURE 3. *Example of String Reference-Based Compression*

Constructing a Universal String Reference

Constructing a universal reference presents an optimization problem over a given universe of genomes. In this paper, we consider bacterial genomes. However, we hypothesize that these methods are transferable to other sets of genomes, regardless of size and features.

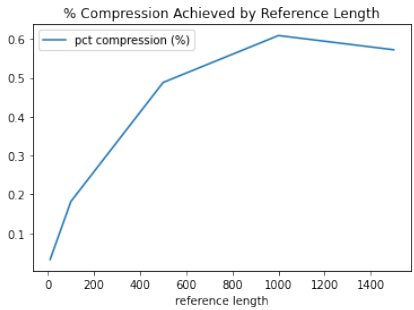
We first devise an exploratory test utilizing Algorithm 1 to determine the time and efficacy of our approach across different reference string sizes. In this test, we arbitrarily use the first N bases from the reference genome `GCF_013417135.1_ASM1341713v1` from `Acinetobacter_sp._YH16051`. Tested efficacy and time-efficiency in constructing the substring trie are elucidated in Figure 4. We observe a slight decrease in performance at a reference length of 1500. This is explained by our linear algorithm that greedily matches substrings with those in the reference. This drop-off in performance is ameliorated by dynamic programming. We observe quadratic growth in time to construct the substring trie, as expected.

Heuristics

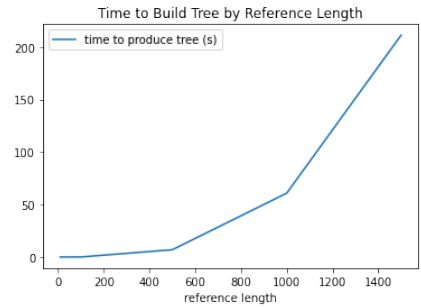
Studying the *10bacteria* dataset (Annex A), it was common to see long contiguous stretches of `N` and `-` characters, averaging ≈ 50 bases. Hence, we can utilize this as a heuristic when constructing our string-based references. We detail a demonstration as follows:

1. Consider our previously-tested string reference, 1000 in length.
2. Replace the last 100 characters with 50 `N` and 50 `-`.

We note the considerable 24.5% uptick in performance (average percent compression) in Figure 5, testing on the *100bacteria* dataset (Annex A).



Percent Compression Achieved by Reference Length



Time Required to Construct Substring Trie

FIGURE 4. *String-based Reference Analysis*

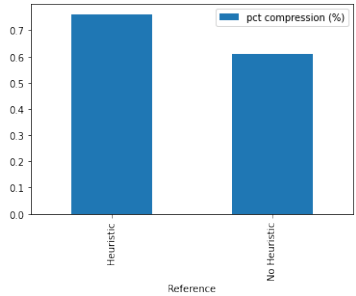


FIGURE 5. *Performance of Reference with and without Heuristic*

A Genetic Algorithm

There exist several methods of optimization; however, we utilize a genetic algorithm for generating such reference strings due to the absence of needing to calculate a gradient (and thus score) for all neighboring strings, as constructing the substring trie is computationally expensive (quadratic).

Algorithm 2: Genetic Algorithm: Searching for an Optimal String Reference

Result: Genetically optimized reference string (not guaranteed to be optimal)
Init N, R, O;
Init M; (Bases' of mutation in offspring)
population ← Randomly generate N + R + O references;
for *e in EPOCHS* **do**
  Get score of each reference;
  population ← N offspring*, R new random references, O best old references from population;
end
return *Best performing reference;*

Our genetic algorithm is defined in Algorithm 2. *Offspring are generated by sequentially sampling two parent references, weighted by score and combining them at some arbitrary index. References are sampled in accordance with the aforementioned heuristic.

Performance results from the genetic algorithm are presented in Figure 6. References are trained on the first 25 bacteria genomes in *100bacteria* (Annex A) and are tested on the remaining 75 bacteria genomes in *100bacteria*. The reference achieving the best compression is presented in Figure 7.

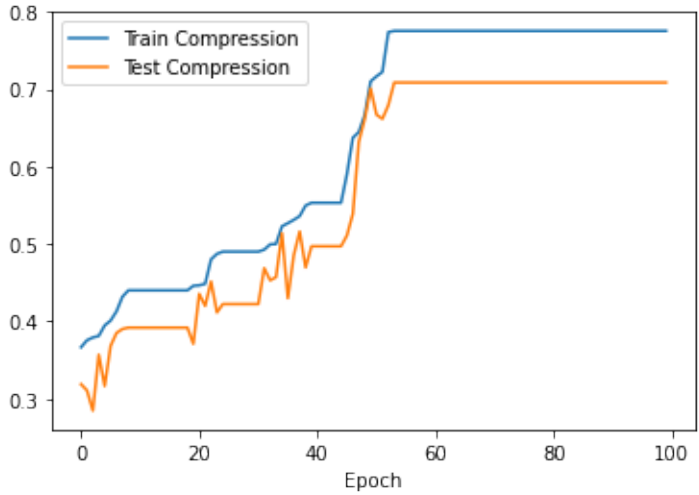


FIGURE 6. *Train and Test Compression of Genetic Algorithm. N=30, R=2, O=2, Reference Length=500*

```
 AGCGCAGGCGGCGCGGATGCCGTCCCGATCGGCCGGCGACGTCGAGGGACTGGGCGGTCCCT-
GGTGACGACCCTGGCCCCGGTGATCGCCGAGGCCGCCCGACGCGACCGCGACCTGGGTGCCCA
CACNGTTCTCGTCGAGGGCGACCGCCCGGGCGATGCGGCGCTGTTCGACGCGGCCGCCGACCA
TCTGGCCGAGGGCGGTCAGTGGCGAGGCCGGTTGCGGCGGTCCCCGCAGCAGCGTGCTGTCGA
GGAGACGCCGGTACGGGCGAGCGTCGACGGCCGGGTCCCGACCGACGAGGCGTCGTTGCGACG
GGTGG-CGACCACCTGGCGGTCGTCGACGCCGTCCATCAGGTCCATCGCAACCTGGCCGACCT
CCATGTTCCGATCGATGCCTCCNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                NNNNNNNNN---------------T--
```

FIGURE 7. *Top Scoring Reference. 0.77% (train) 0.68% (test) compression, benchmarked against 3-bit encoding.*

Suggested Improvements to Reference String Approach

Similar to suggested improvements to the *Reference Dictionary Approach*, more computational power is required to completely appreciate the potential of the string reference. Greater computational power allows the use of longer reference strings and dynamic programming to find the optimal compression of genomes rather than the greedily assigned compression detailed in this method.

Conclusion:

While great success has been found utilizing reference compression for compressing groups of similar genomes such as multiple alignments, effective performance was not realized when applied to individual genomes. While the proposed improvements might improve the performance of this method, it seems unlikely that we will be able to achieve any meaningful improvement on the 75% compression rate offered by a 2 or 3 bit encoding.

Section 2: The Compression of Large Datasets of Similar Genomic Information

Introduction:

Much of the genomic data people operate on comes in massive repositories. For example, to study the evolution of COVID across time and geographic regions it isn't sufficient to analyse a single COVID genome. Rather, one often needs hundreds, sometimes thousands. To correlate genomic and phenotypic variation you similarly need large numbers of different but largely similar genomes. When studying genes one typically obtains thousands of samples. In short, while in conceptual discussions we often discuss a single genome—often a reference genome—in practice biologists often work with the storage, transfer, and manipulation of thousands of genomes.

These large datasets pose a number of challenges, due to how large the file sizes can become. These problems are only set to grow over the coming decades, driven by the exponential decreases in sequencing costs and the associated explosion in the number of genomes sequenced each year (Illumina 2020). A human genome stored in uncompressed format requires 3GB. In uncompressed form, storing a modest 100 human genomes would then require 300GB.

On the one hand, there are difficulties in performing analysis on data of this magnitude. While there are supercomputers with this quantity of RAM, it far exceeds the typical 8-16GB of RAM an ordinary computer might have forcing the computer to resort to slow disk memory, which then prohibits the average biologist from performing real time experiments on this data. Moreover, merely storing this data becomes expensive. Based on a back of the envelope calculation, we estimate that storing all of the data that Illumina has sequenced in uncompressed form would cost upwards of $100 million.

Fortunately, we have come up with an algorithm which helps address these issues.

The rest of this section is structured as follows:

- We begin by motivating the algorithm.
- We discuss some of the advantages inherent in our approach.
- We proceed to describe the algorithm.
- We present data on how our algorithm performed on the 85,000 COVID genome dataset hosted by the NCBI.
- We describe the current limitations and plans for how to address them, as well as future improvements and research directions.

Note: While we did come up with these ideas, we are under no illusion that all or even any of them are novel. We did not do a literature review before diving in and, while we have since looked up surveys to compare how our approach compares to the current state of the art, we did not actually explore that state of the art.

Motivation:

With regards to compression, there are certain advantages which you have when working with many versions of the same genome, which is most clearly summarized in Figure 8. Imagine that each line was a separate genome. If you wanted to communicate this information you would not say,

AAAAAAAAAAAAAAAAAAAAAAAAAAAA\nAAAAAAAAAAAAAAAGAA
AAAAAAAAAA\nAAAAAACAAAAAAAAAAAAAAAAAAAAA\nAAAAAA
AAAAAAAAAAAAAAATAAAAAA\nAAAAAAAAAAAAAAAAAAAAATAA
AAAA

Instead you would articulate

- *There are 28 A's in the first row*
- *The second, third, fourth and fifth rows are identical EXCEPT*
  - *Row 1: There is a G in position 16*
  - *Row 2: There is a C in position 7*
  - *Row 3: There is a T in position 22*
  - *Row 4: There is a T in position 22*

Figure 8 presents a slightly simplified view when compared to real genomic data. For one, in genomic data it is rare to observe long runs.

For another, while our grid contains only the standard 4 genomic letters, ACTG, actual sequencing outputs also contain the less familiar letters [W, S, M, K, R, Y, B, D, H, V, N, Z]. The familiar N represents any base, W represents an ambiguous A or T, and so on. Perhaps the most important difference however is actually advantageous when it comes to compression. Unlike in our Figure 8 where the mutation rate is 1/28 (using the first row as the reference), in actual genomic data it is closer to 1 in 1000. We will discuss all of these factors in greater detail when we present our results.

This idea forms the basis for our algorithm. Instead of transmitting a thousand genomes which are almost indistinguishable, one after the other, requiring space

AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAGAAAAAAAAAAAAA
AAAAAACAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAATAAAAAA
AAAAAAAAAAAAAAAAAAAAATAAAAAA

FIGURE 8. *Large quantities of redundant information with small differences between lines.*

O(number of genomes * length of genomes), we instead first transmit a single reference, and then secondly, for all of the remaining genomes, we transmit not the genome itself but rather the differences to the reference, which losslessly compresses it. Doing this in then reduces space to O(length of reference + (number of genomes * magnitude of differences)). 'Magnitude of differences' is an as of yet underspecified factor, but, as we will see, it is tiny when compared to the length of the initial genome.

Advantages:

The advantage when it comes to storage is obvious. When one has many genomes which are similar to one another, with long regions of high similarity, storing the differences between the strings is likely far more economical than storing the whole string. However, there is another advantage which is hopefully also evident. If we gave you the long naive string and asked you to tell us something about it you would be stumped. You would first have to align them, then somehow search for areas of interest, and so on. On the other hand, if we gave you the data in this second format, you could immediately tell us that:

"Mutations have occurred in base positions 16, 7, 22. Interestingly, 50% of our sequences contain that mutation in position 22. Perhaps it is an evolutionarily advantageous mutation?"

In short, storing the data in this format immediately draws your attention to the regions of interest. If there is a 10,000 BP region of DNA which is the exact same across all of your organisms, you never look there, as you would otherwise be required to do. Instead you get all of the information you are interested in directly, neatly summarized - information regarding point mutations and where indels have occurred. Having compressed billions of data points down to (as will be seen) orders of magnitude fewer, it becomes feasible to study the data visually and make intuitive/human observations, rather than having to rely on algorithmic processing and aggregations.

While obvious in retrospect, this was a somewhat unexpected positive externality. To provide an anecdote. When we compressed 85,000 COVID genomes (as will be discussed later) we saw the same numbers appearing again and again. Without any analysis whatsoever we identified regions which appeared to be interesting, which we later verified as emerging mutations which were becoming increasingly prevalent across time.

Algorithm:

Algorithm 3: Differential Compression of Genomes

Result: A binary file encoding sequences of reversible transformations which translates each of the initial raw genomes into the reference.

1) Select or create a reference.
for *each genome G in your dataset:* **do**
    2) Pairwise align G with your reference.
    3) Identify a series of transformations which produces G starting from the reference.
end
4) Once you have obtained this differential data, you need to encode it in binary bits.
return *A binary file.*

We will discuss each of these steps individually.

Step 1: Selecting a reference

At first we considered trying to construct a "modal" genome representing our dataset. By this we mean a genome which is in some sense the average of all of the other genomes - for example, having the modal value for each base position. The limitations to simply choosing an arbitrary genome should be quite evident. Suppose for instance that we selected the first ever full COVID genome as our reference. Overtime new mutations occur. Suppose that the 1001th genome that was sequenced mutated base X to another letter, and that this mutation was so advantageous that all of the 84,000 genomes after this one have the mutation in base X. If the reference contains the original base you will have to encode this deviation 84,000 times. If you instead encode the mutated version in your reference, you only need to encode 1000 deviations. This constitutes a hundred fold difference in your compression.

Unfortunately, figuring out what the 'modal' reference should be is not a trivial task. When we began our analysis we figured using the NCBI standard reference would be a sufficiently good starting point, and it turned out to perform fantastically well. We re-ran our compression algorithm using other randomly selected genomes and these performed similarly. These results suggest that in practice, whether or not you have a reference genome available to you, arbitrarily selecting any you have available (assuming it is complete and does not have long gaps) will perform well. It is important that it is complete because if it isn't the information about any gap will have to be encoded in full in the other genomes (rather than small differences).

While unverified, we have a sense for why. Large portions of the genome are very conserved and remain the same across time. Mutations do occur but most are relatively infrequent and occur in only a small proportion of genomes. There are a few mutations which, as described, are so advantageous so as to confer a sufficiently

large fitness advantage that they become significantly more frequent across time, but these are sufficiently rare that, even if you do the suboptimal thing of encoding 84,000 instead of 1000, your overall compression is still highly effective.

We have a relatively clear idea of how we would go about producing a modal reference (unfortunately time limitations prohibited us from doing so) and do believe that it would result in improved compression ratio. That said, we do not believe that this is where improvements would yield the most significant impact. We will discuss the limitations of our approach in a later section, as well as briefly discuss what improvements we know could be made, as well as changes we theorize might be advantageous.

Step 2: Pairwise alignment

For this step we used off-the-shelf software implemented in the software package Biopython. We used the PairwiseAligner method, scoring matrix NUC.4.4 (also known as DNAfull), gap opening penalty of -10, and gap extension penalty of -2.

```
aligner = Align.PairwiseAligner()
aligner.open_gap_score = -10
aligner.extend_gap_score = -2
aligner.substitution_matrix = substitution_matrices.load("NUC.4.4")
```

The gap extension penalty has to be equal to or higher than the ACTG-N mismatch score, so that the program prefers this type of mismatch to long gaps.

We did not do an extensive parameter search. The mentioned values performed well in practice.

Step 3: Identifying a series of transformations

Figure 9 illustrates a sample procedure.

1. First, we align the reference and G.
2. Next, we store at what positions we should insert characters relative to the reference.
3. We proceed to store where deletions have occurred, and how long they are.
4. We replace long runs of Ns with what the reference contains.
5. We identify the positions of non-ACTG letters.
6. Finally we store the location and character of mismatches.

A few natural questions arise looking at this procedure. One may ask why N's and other letters are treated differently. The reason is that Ns typically occur in very long runs, while the other characters are relatively rare and typically occur individually. We do not want to have to store (100, N), (101, N), (102, N), (103, N) ..... - this is done much more concisely by "(starting position, length)". Mismatches and abnormal letters are also treated differently. This is done so that when you later encode your mismatches you can do so using just 2 bits to indicate A, C, T, G.

Note that the order of these different operations is not critical. That said, different orderings do have different pros and cons with regards to later encoding, and have different edge cases to consider.

For an example of the output of our differential algorithm on real data, see Figure 10 which displays the differential representations of 10 COVID genomes.

Step 4: Encoding the transformations

One of the most important steps in the procedure and the one which, being the least interesting, we unfortunately devoted the least amount of time to. The goal is to use as few bits as possible to encode each piece of information, while maintaining the condition that your encoding has an inverse. Our current implementation is suboptimal - we optimized for software development speed.

A few quick notes:

One wants to pare down on non-essentials to the greatest extent possible. No transitions, no termination characters, etc. You accomplish this by defining a very specific scheme, so that when you want to decompress your data you can do so unambiguously. To make things concrete, suppose you wanted to store (15, "A"). If you knew ahead of time that you stored numbers using 10 bits and the subsequent letter using 2 (with a defined pairing e.g. 00: "A", 01: "C", 10: "T", 11: "G") you can encode a long list of these using just 12 bits for each. If you needed to know when you switch to discussing Ns instead of mismatches, you merely prefix this block of 0s and 1s with an integer (which again you store using a specific number of bits) indicating how many of these pairs there are. Then you read the first x bits to get the number of pairs, and then blocks of 12 x number of times, before switching over to interpreting the bits in some other manner.

Results:

To test the performance of our compression scheme, we downloaded the 85,000 COVID genomes hosted on NCBI. We losslessly compressed the genetic data from 2.57GB to 4.61MB, which translates to a 558x compression factor. Put another way, the marginal cost of adding another genome to the dataset decreased from 30,000 to 55 bytes.

While we didn't do a comprehensive analysis, our compression generalizes and performs similarly on other datasets. To get a sense for how much space you might expect X genomes of length L to require, use the equation

$$L/4 + X * L/500$$

with the terms representing the cost of storing a reference (the factor of 4 is due to reducing the space per base from 8 bits (ASCII) to 2, and the second is an estimate for the marginal cost of storing another genome, based on the datasets that we have compressed.

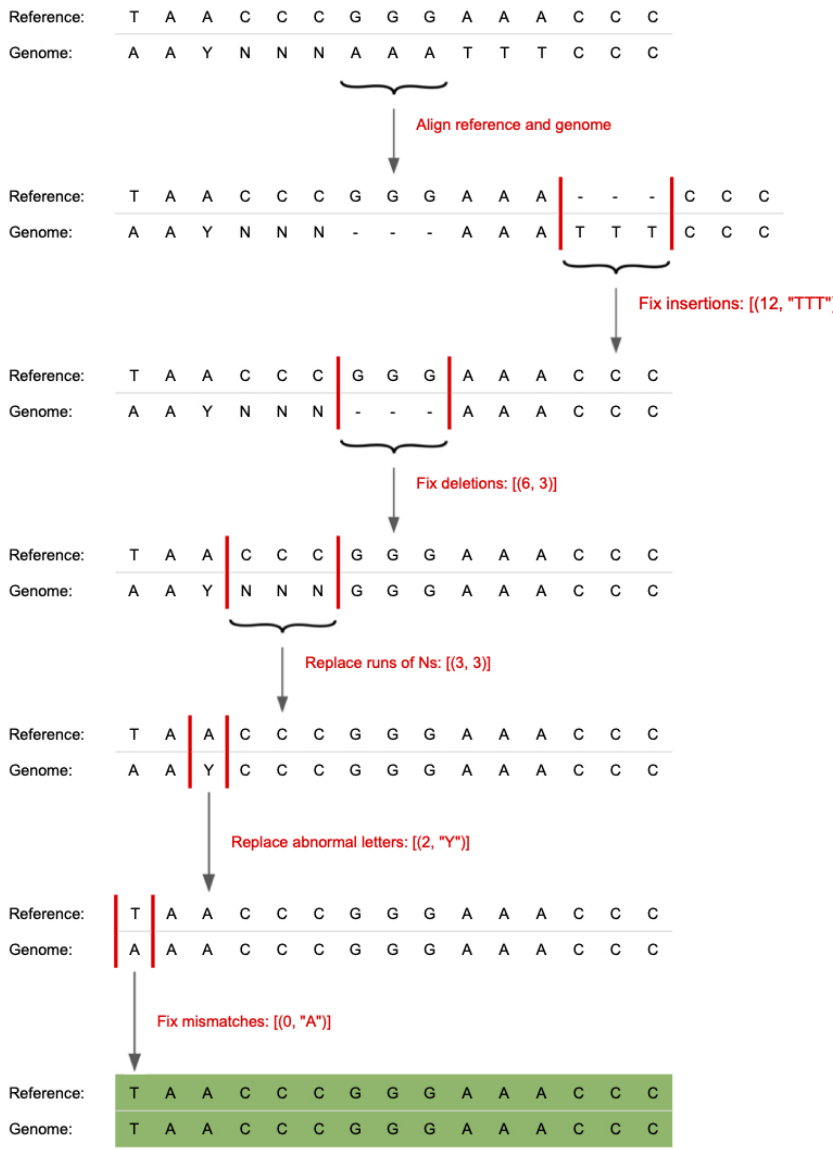


FIGURE 9. *Differential Compression Example*

([],[(0, 54), (29836, 67)],[(19275, 295)],[(3036, 'Y'),(4330, 'Y'),(8781, 'Y'),(9476, 'W'),(13729, 'Y'),(14407, 'Y'),(14804, 'Y'),(25428, 'K'),(25978, 'K'),(28143, 'Y'),(28656, 'Y'),(28862, 'Y')],[])

([],[(0, 54), (29836, 67)],[],[(240, 'Y'),(3036, 'Y'),(4539, 'Y'),(8781, 'Y'),(9476, 'W'),(9855, 'K'),(12241, 'Y'),(14407, 'Y'),(14804, 'Y'),(18451, 'Y'),(22878, 'M'),(23402, 'R'),(25978, 'K'),(26028, 'M'),(27753, 'R'),(28143, 'Y'),(28656, 'Y'),(28862, 'Y')],[])

([],[(0, 54), (29686, 67)],[(2568, 135),(2706, 144),(7092, 240),(10737, 285),(16485, 127),(16616, 17),(16636, 116),(19275, 295),(20231, 265),(21146, 240),(27511, 297),(29665, 2),(29753, 150)],[],[(489, 'A'),(3176, 'T'),(8781, 'T'),(18735, 'C'),(19683, 'T'),(24033, 'T'),(26728, 'C'),(26780, 'T'),(28076, 'C'),(28143, 'C')])

([],[(0, 54), (29836, 67)],[(19275, 121), (19404, 11)],[(240, 'Y'),(3036, 'Y'),(7990, 'Y'),(8130, 'K'),(8781, 'Y'),(12069, 'K'),(14407, 'Y'),(17125, 'Y'),(17469, 'Y'),(20267, 'R'),(22334, 'K'),(23402, 'R'),(26087, 'Y'),(27577, 'W'),(28143, 'Y')],[])

([],[],[(0, 54),(4995, 292),(6846, 212),(19275, 295),(19911, 289),(21146, 240),(27511, 593),(29836, 67)],[],[(921, 'A'),(3989, 'T'),(8781, 'T'),(11229, 'T'),(23402, 'G'),(25848, 'A'),(26207, 'T'),(28143, 'C'),(28166, 'A'),(28877, 'A'),(29741, 'A')])

([], [], [], [], [(1996, 'T'), (8781, 'T'), (27502, 'T'), (28143, 'C')])

([],[(0, 15)],[],[],[(4401, 'C'), (5061, 'T'), (8781, 'T'), (28143, 'C'), (29741, 'T')])

([],[(0, 187), (3045, 26857)],[],[],[(240, 'T'), (1058, 'T'), (3036, 'T')])

([],[(0, 43), (29170, 74)],[(43, 358),(443, 309),(814, 295),(1148, 279),(1474, 314),(1842, 705),(2720, 56),(2899, 626),(3564, 256),(4212, 299),(4580, 281),(5307, 317),(5656, 278),(5980, 293),(6326, 588),(6967, 322),(7972, 306),(8332, 292),(8663, 257),(9283, 256),(9615, 264),(9923, 265),(11486, 229),(12132, 2688),(14847, 292),(15191, 302),(15499, 267),(15800, 691),(16504, 2),(16534, 246),(17134, 671),(17859, 262),(18564, 832),(19516, 247),(19812, 263),(20425, 635),(21111, 237),(21403, 617),(22023, 672),(22747, 291),(23092, 332),(23468, 310),(23822, 275),(24150, 292),(24484, 284),(24820, 285),(25178, 3270),(28492, 262),(29246, 657)],[],[(28880, 'A'), (28881, 'A'), (28882, 'C')])

([],[(0, 43),(1813, 1),(3412, 1),(5224, 1),(8464, 1),(8905, 1),(9627, 1),(9811, 1),(11074, 1),(13914, 1),(16630, 1),(19983, 1),(24672, 1),(25100, 1),(25587, 1),(26561, 1),(29377, 1),(29829, 74)],[],[],[(8781, 'T'), (16380, 'A'), (28143, 'C'), (28877, 'A'), (29741, 'A')])

FIGURE 10. *Differential Representation of 10 COVID Genomes*

Unfortunately due to time constraints we were unable to do a proper profile analysis of memory consumption (i.e. whether it was mismatches or runs of Ns which consumed the most space). That said, due to cutting some corners we are aware of a number of inefficiencies which could individually be fixed given an hour or two. We will be discussing these later.

We did profile the runtime of the compressor. Unfortunately, the compression step itself was quite slow, taking approximately 15 seconds per genome (139 single core CPU hours / GB). Fortunately the process is identical for each genome and so is easily parallelizable, and we took advantage of the 40 core Andrew machines to compress the dataset in a reasonable time.

The breakdown of runtime costs was unsurprisingly dominated by the alignment step, which accounted for over 99% of the cost. All of the other steps in the process are linear, while the Python PairwiseAligner() function we used was quadratic in the length of the genome, using the dynamic programming approach that we have covered in class.

Limitations, Improvements, and Future Research:

Improving runtime:

Most pressing is reducing the runtime cost to a reasonable level. Using zip for example the initial 2.58GB file is compressed in under a minute. We are wholly convinced that similar performance using this approach is possible. We believe that we can reduce the alignment step to linear time by using a seed and extend approach rather than the quadratic dynamic programming approach. We did not have time to attempt to implement such an algorithm (and were unable to find any existing software packages which ran pairwise alignment or approximations thereof in linear time) but believe that such an approach would definitely be possible, perhaps at the sacrifice of guarantees of optimality. That said, for this approach to work one merely needs something good, not perfect. [1]

Improving memory:

This project ended up requiring a lot of time, and so certain aspects were rushed. Many of the corners which were cut are simple fixes - other things which we know/believe would improve performance are more complex.

- Encodings: We did not devote much time to the encoding of the information. For example, instead of encoding characters ACTG using just 2 bits we resorted to ASCII for the sake of simplicity. Some of our numerical information was similarly

1. I, Edward, plan to attempt to implement a linear time aligner once the semester ends and I have some more time. I think fast alignment would be a useful tool.

encoded in a blatantly ineffective manner. We stored information like (121, "Y"), (4242, "Y"), (. . . , Y) as a simple list instead of doing something more intelligent like "Number of Y mismatches | Position1, Position2, Position3" – saving the need to repeat the encoding of Y. A few quick napkin calculations suggests that fixing some of these issues would yield improvements upwards of 50%.

- Frequency analysis: As one discovers looking at the informational blocks we derived from the genomes, mutations and changes do not occur randomly distributed throughout the genome. Some mutations occur very often. Consider Figure 10. Figure 3 is the code representation of the inferred transformations of the first 10 genomes. Just looking at the data it becomes apparent that some numbers occur very often. For example, 8 of the 10 genomes has a mutation at base 28143. Doing a frequency analysis on just these 10 genomes one learns that the 10% most frequent numbers account for 30% of the overall numbers (and these numbers become more extreme the more genomes you have). Given this, it makes no sense to treat all numbers equally. Instead one can use a Huffman encoding to optimally encode this data (Huffman 1952).

  Note also that this (to an extent) attenuates the issues mentioned in Section Algorithm, Step 1, with regards to choosing a suboptimal reference. The example given there was choosing a reference which has letter X in position Y, while in reality 99% of the genomes actually have another letter. This mistake becomes costly since you need to store the same deviation many times. However, if you take into account how often a particular number occurs as described above, this becomes less of an issue since any number which occurs very often will be encoded using fewer bits.
- The order of transformations: As noted in a previous section, what order you perform the transformations in does matter for compression. The order changed a few times while we were developing the software – primarily to reduce the number of edge cases which had to be considered – and we observed that ordering did make a non-trivial difference (assuming you were willing to take some time and take advantage in the encoding phase) but did not have time to do a more thorough investigation.
- Suboptimal reference: While it is not immediately obvious how to do so, we believe that, given the pairwise alignments, we could create a better reference by overlaying the alignments, and selecting 'modal' values. While attempting to do any sort of multiple alignment would be far too costly, we believe that 90% of the same benefit can be reaped through a simpler and less costly approach.

Overall, based on some rough analysis and experiments we ran, we believe that improving the encoding scheme, as well as using Huffman encoding to take frequency information into account when encoding numerical information, we could improve the compression ratio by upwards of 2x, which if realized would yield 1000x+ overall compression.

Improving the reference used would have been a significantly more involved project. Not having attempted to do so, we don't have a good sense for what sort of an

impact this would have. While it would certainly not be insignificant (for example, using a more intelligent modal value for position 28143 would have resulted in far fewer mismatches and thus bits) we don't know whether these suboptimal choices occurred frequently enough to make any material difference.

Extensions:

We applied our compression scheme specifically to datasets containing many genomes from the same species. However, there is nothing prohibiting this approach being used more generally. While compressing a file containing haemoglobin molecules from 100 different species will undoubtedly perform worse than performing a similar compression on a file of 100 human haemoglobin sequences (since there is more variation between species than within one, there will be more differences to store) we suspect that compared to standard compression techniques this differential approach will still compare favorably.

One can compress files containing reads using a similar technique. You can either search through a database for matches or directly look up a reference from the species your reads come from. Then, instead of storing the reads verbatim you can encode the position the read appears in the genome, along with information about any mismatches / differences which occur.

Conclusion:

It is clear that, at least in certain cases, one can compress genomic datasets by amounts which are, by any normal standard, somewhat outlandish — speaking to the redundancy the raw files contain. Our work suggests that, assuming this technique can be made faster, in many cases it can be very profitable to compress one's datasets, enabling faster transfer, cheaper storage and, as we have seen, sometimes easier analysis.

Section 3: The Application of Algorithms to Compressed Data

Introduction:

The increasing size of biological datasets imposes higher costs not only on data storage, but also on its analysis. Given the limits on the capabilities of the average computer, as well as the fact that in most cases there is a high redundancy in the biological data that needs to be analyzed, we believe that applying analysis directly on compressed versions of data rather than on their raw form is an idea with high potential impact.

Understanding the challenges associated with performing analysis on compressed data is important when considering different compression schemes, since it is important that the design of compression schemes facilitates rather than hinders the performance of algorithms that might be applied.

For example, the approach for compressing multiple genomes discussed in the previous section lends itself to analysis of the frequency of different mutations over time. The approach also might be used to directly produce a multiple alignment. Since it is easy to see from the compressed data where genomes agree with each other and where they differ, the compressed data on its own offers important clues that can help to produce a high-scoring multiple alignment.

In the remainder of this section, we will examine in detail one idea for performing analysis on compressed data: the application of the Burrows-Wheeler Transform based pattern matching algorithm to a run-length-encoded version of the BWT string.

As will be seen, while a natural idea, applying run length encoding to the BWT string did not in fact offer any compression advantages. Nonetheless, while it turned out to be somewhat contrived, this attempt was nonetheless useful in learning about the challenges associated with applying algorithms directly to compressed data.

Compression Using the Burrows-Wheeler Transform with Run-Length Encoding

Compression and Decompression

The Burrows-Wheeler transform (BWT) of a nucleotide sequence (Burrows and Wheeler 1994) forms the basis for an efficient pattern matching algorithm (Li and Durbin 2009). The BWT has a useful property: it tends to transform strings containing many repeats into strings containing many runs. The logic behind this phenomenon is illustrated in Figure 11.

This property leads naturally to a compression approach that combines the BWT with the common data compression strategy of run-length encoding (RLE). The defining advantage of such an approach is that the BWT-based pattern matching algorithm can be applied directly to the compressed data, without the need for decompression.

Our RLE strategy is illustrated in Figure 12. Each nucleotide is represented using two bits. A string of two of the same character is used to indicate a run. The bits immediately after two identical characters represent how many additional characters

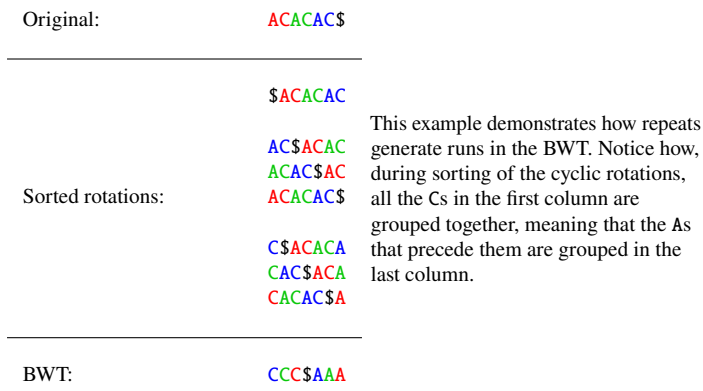


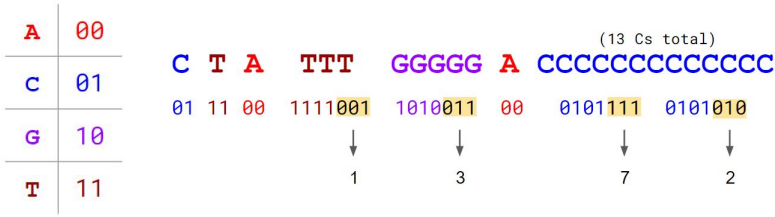
FIGURE 11. *The Burrows-Wheeler Transform and Repeats*

are in the run. The number of bits used for this can be chosen depending on the characteristics of the data; data with more, longer runs would benefit from more such bits (since longer runs can be represented), while data with fewer, shorter runs would benefit from fewer such bits (to decrease the extra bits necessary to represent shorter runs). The example in Figure 12 uses 3 bits. After analyzing the characteristics of the distribution of run lengths in the BWTs of a few test datasets, the final choice was to use a single bit to represent run lengths. The reasoning behind this will be discussed further later.

Pattern Matching

The BWT pattern matching algorithm described by Li et al. can be performed directly on this compressed data, without the extra memory cost associated with decompressing the data. The details of this algorithm will not be discussed here. However, there are a few pieces of information that cannot be determined directly from the compressed data that are required in order to perform the algorithm.

First of all, in order for the algorithm to run, it must be possible to locate a character in the BWT given its index. It must also be possible to determine which occurrence each character in the BWT is. Since runs are condensed, these things cannot be inferrred from the compressed data with just a character's index. For this reason, a checkpointed Counts array must be constructed, which will include both the character counts at certain positions ("checkpoints") as well as pointers into the compressed data that indicate the locations of characters at the indices corresponding to the checkpoints. This checkpointed Counts array can be generated with one pass through the compressed data. The frequency with which it stores checkpoints is controlled by the parameter `C`—the lower it is, the more frequent the checkpoints will be, and the faster the pattern matching algorithm will be able to determine the



This diagram illustrates our RLE strategy. Two consecutive characters indicate that a run is starting. The following bits represent the number of additional characters in the run.

This example uses 3 bits to represent run lengths. Notice that because of this, the representation of the run containing 3 Ts is longer than it would have been if RLE was not used. On the other hand, the run of 13 Cs must be split up, since it exceeds the longest representable run with 3 bits. This demonstrates the tradeoff that must be made when choosing how many bits to use to represent run lengths.

FIGURE 12. *RLE Compression Strategy*

locations and occurrences of characters in the compressed BWT. However, more checkpoints will require more memory to store. Thus, the choice of `C` determines how the algorithm will handle the tradeoff between speed and memory usage.

Analysis

As purely a compression strategy, this approach is less than ideal. As can be seen in Figure 13, where it was tested on the first half million `ACGT` bases of various organisms' genomes, it performs consistently worse than the 75% compression ratio that would be expected from using two bits to encode each base.

The reason for this has to do with our RLE strategy. Since runs are represented by two of the same character followed by a predetermined number of bits representing run length, runs of length 2 will always require more space than they would if two-bit encoding was used alone. Figure 14 demonstrates the unfortunate fact that the vast majority of the runs in the BWTs of the test datasets had length 2. This means that the benefits of compressing the longer runs were nullified by the disadvantages of compressing shorter runs for these organisms, even using just 1 bit to represent run lengths.

Despite this, the advantage of this compression strategy is the ability to perform the pattern matching algorithm directly on the compressed data. In the Pattern Matching section, we describe the parameter `C`, which determines the frequency of checkpoints in the Counts array. Another important parameter is `K`, which is the frequency of checkpoints in the checkpointed Suffix array. These parameters affect the time-memory tradeoff of the algorithm. Figure 15 shows how the time that the algorithm takes to run varies with `K` and `C`. The easy editing of these variables offers

	Burrows-Wheeler Transform + RLE	RLE Alone
Plant (*Arabidopsis thaliana* chr. 2)	Source length: 500000 Compressed length: 128510 Compression ratio: 74.30%	Source length: 500000 Compressed length: 129825 Compression ratio: 74.04%
Archaea (*Halobacterium salinarum*)	Source length: 500000 Compressed length: 128484 Compression ratio: 74.30%	Source length: 500000 Compressed length: 131817 Compression ratio: 73.64%
Human (*Homo sapiens* chr. 21)	Source length: 500000 Compressed length: 125578 Compression ratio: 74.88%	Source length: 500000 Compressed length: 128856 Compression ratio: 74.23%
Virus (*SARS-CoV-2*)	Source length: 30429 Compressed length: 7785 Compression ratio: 74.42%	Source length: 30429 Compressed length: 7833 Compression ratio: 74.26%
Bacteria (*Staphylococcus aureus*)	Source length: 500000 Compressed length: 129547 Compression ratio: 74.09%	Source length: 500000 Compressed length: 130102 Compression ratio: 73.98%

File sizes are in bytes. Non-ACGT characters were ignored (for the larger files, they were removed before truncating the file to the first half million characters).

FIGURE 13. *BWT with RLE Compression Statistics*

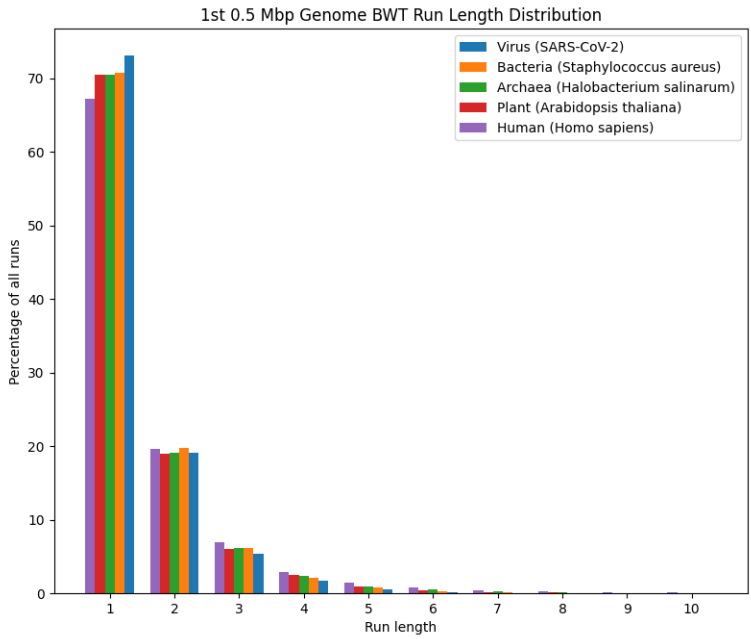


FIGURE 14. *Distribution of BWT Run Lengths in Test Datasets*

flexibility to users, allowing them to adapt the algorithm to reduce the computational costs that they might prioritize.

		K			
		10	100	1000	5000
C	10	0.129 s	0.102 s	0.144 s	0.346 s
	100	0.112 s	0.198 s	1.162 s	5.399 s
	1000	0.220 s	0.963 s	8.350 s	40.685 s
	5000	0.862 s	5.967 s	55.389 s	271.596 s

The parameters `K` and `C` determine the frequency of checkpoints in the Suffix and Counts arrays, respectively, that are used by the BWT-based pattern matching algorithm. This table shows the time, in seconds, taken to run the matching algorithm with different values of `K` and `C` on a compressed file generated from the NCBI reference SARS-CoV-2 genome, whose NCBI accession number is `NC_045512.2`. The pattern that was searched for was `CTAAAAA`, which occurs ten times in the SARS-CoV-2 reference genome, at locations 1811, 3410, 4172, 5222, 8462, 13584, 16798, 21064, 23141, and 29375 (indexing by zero, ignoring newlines).

FIGURE 15. *Checkpoint Parameters' Effect on Matcher Speed*

Challenges

Some obstacles that were encountered in the course of implementing the pattern matching algorithm are detailed below.

1. Because our compression scheme condensed runs, the index of any given character in the compressed data could not be calculated directly. If we wanted to know where the 351st character was in a non-compressed file, we could quickly access it; run-length encoded files do not allow for this. Thus, auxiliary data was required (in the form of the Counts array with pointers into the compressed data) in order to locate any character, whose generation incurred efficiency costs.
2. The checkpointed Suffix array, which was required for finding the indices of cyclic rotations, could not be produced from the data without essentially decompressing it. For this reason, it had to be stored in the footer of the compressed data.
3. Appending an explicit `$` character to the input sequence is necessary in order for the BWT to be invertible for all inputs. However, choosing an explicit encoding for this character would increase the alphabet size beyond the four nucleotide bases, making it impossible to base our compression scheme on two-bit encoding. This was resolved by removing the `$` before compression, and storing the location from which it was removed at the end of the compressed file. This substantially complicated the application of the pattern matching algorithm to the data, since any time the compressed data was read, the possibility that `$` was inserted at that location needed to be handled.

These issues reveal fundamental difficulties with applying analysis to compressed data in general. Depending on the chosen compression scheme, the transformations performed in the compression of the data might result in a loss of valuable information that is necessary for performing analysis. For this reason, it is crucial that the

compression technique is compatible with analysis. Bad engineering of compression schemes will make analysis more difficult, while good engineering has the potential to make analysis easier. This is an important consideration to keep in mind when evaluating different ways of compressing data.

Conclusion:

Our treatment of the BWT/RLE compression strategy in this section has revealed that applying this particular run-length encoding scheme to BWTs failed to be an effective or useful compression strategy. Despite this, our examination of the application of the BWT-based pattern matching algorithm to the compressed data demonstrated the inherent problems that arise when working with compressed data. The considerations raised in this section can inform future efforts in adapting compression strategies to analysis, and in working with compressed data directly.

Conclusion

The genomic data compression approaches that we have explored throughout this paper have had varied results. However, they have all exposed key considerations to take into account in further research.

Our investigation into universal dictionary and string-based references exposed difficulties in constructing references that are sufficiently general so as to be effective in the compression of arbitrary data. We ultimately concluded that attempts to compress arbitrary data using a universal reference would not be able to achieve compression ratios comparable to those obtained by using more specialized references, due to inherent limitations imposed by the diversity of genomic data.

Our differential compression algorithm demonstrated state of the art level performance on the COVID-19 genomic dataset hosted by the NCBI, achieving compression rates above 500x. This suggests that large datasets may be very fruitfully compressed.

Finally, our discussion of applying analysis directly to compressed data revealed the importance of crafting compression schemes that are amenable to direct analysis. Our discussion of using the Burrows-Wheeler Transform with run-length encoding as a compression approach provided a concrete example of how these difficulties might come into play.

The immense rate at which biological information is being collected demands sustainable mechanisms for its storage and manipulation. We hope that the approaches we have explored in this paper, as well as the observations that were presented, can serve as a solid starting point for tackling this information deluge.

References

Boufounos, Petros, Sameh El-difrawy, and Dan Ehrlich. 2002. Hidden Markov Models for DNA Sequencing (October).

Burrows, Michael, and David Wheeler. 1994. A block-sorting lossless data compression algorithm. Paper presented at the Digital SRC Research Report. Citeseer.

Deorowicz, S, A Danek, and M Niemiec. 2015. GDC 2: Compression of large collections of genomes. *Sci Rep* 5 (11565).

GenBank, NCBI. 2021. GenBank Overview, May. Available at <https://www.ncbi.nlm.nih.gov/genbank/>.

Huang, ZA, Wen Z, Deng Q, Chu Y, Sun Y, and Zhu Z. 2017. LW-FQZip 2: a parallelized reference-based compression of FASTQ files. *BMC Bioinformatics* 20 (18):179.

Huffman, David A. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40 (9):1098–1101.

Illumina. 2020. Illumina Acquires Enancio's Compression Software, June. Available at <https://www.illumina.com/company/news-center/feature-articles/illumina-acquires-enancio-s-compression-software.html>.

Kozanitis, C, C Saunders, S Kruglyak, V Bafna, and G Varghese. 2011. Compressing genomic sequence fragments using SlimGene. *Journal of computational biology : a journal of computational molecular cell biology* 18 (3):401–413.

Li, Heng, and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *bioinformatics* 25 (14):1754–1760.

RefSeq, NCBI. 2021. RefSeq: NCBI Reference Sequence Database, May. Available at <https://www.ncbi.nlm.nih.gov/refseq/>.

Sulistyawan, I Gede Eka, Achmad Arifin, and Muhammad Hilman Fatoni. 2020. An Adaptive BWT-HMM-based Lossless Compression System for Genomic Data. Paper presented at the 2020 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM), 429–434. https://doi.org/10.1109/CENIM51130.2020.9297871.

Annex A: Datasets

Dataset 1: 10bacteria

Species	NCBI RefSeq Genome Name
Gryllotalpicola_ginsengisoli	GCF_000419445.1_ASM41944v1
Eubacterium_oxidoreducens	GCF_900104415.1_IMG-taxon_25933392
Guyparkeria_sp._SCN-R1	GCF_003932495.1_ASM393249v1
Akkermansia_sp._BIOML-A11	GCF_008422405.1_ASM842240v1
Enterobacter_sp._MGH_15	GCF_000534555.1_Ente_cloa_complex_
Kocuria_polaris	GCF_000786195.1_ASM78619v1
Brasilonema_octagenarum	GCF_012912125.1_ASM1291212v1
Imperialibacter_sp._EC-SDR9	GCF_902498805.1_IMPR6_assembly
Cupriavidus_sp._SW-Y-13	GCF_009793345.1_ASM979334v1
Fusobacterium_gonidiaformans	GCF_000158835.1_ASM15883v1

Dataset 2: 100bacteria

Species	NCBI RefSeq Genome Name
Exiguobacterium_alkaliphilum	GCF_000714435.1_Exiguobacterium_al
Clostridium_sp._AF23-6LB	GCF_003461085.1_ASM346108v1
Corynebacterium_sp._320	GCF_008830455.1_ASM883045v1
Eikenella_sp._NML96-A-049	GCF_001648495.1_ASM164849v1
Actibacterium_naphthalenivorans	GCF_014196965.1_ASM1419696v1
Enterobacter_sp._10-1	GCF_002270295.1_ASM227029v1
Dongia_mobilis	GCF_004363235.1_ASM436323v1
Arcobacter_cibarius	GCF_013372265.1_ASM1337226v1
Blastococcus_aggregatus	GCF_900221005.1_IMG-taxon_2734482
Enterobacter_sp._SM1	GCF_009811955.1_ASM981195v1
Aliivibrio_sp._EL58	GCF_900312675.1_Aliivibrio_sp._str
Citrobacter_sp._wls758	GCF_005280915.1_ASM528091v1
Labrenzia_alexandrii	GCF_000158095.2_ASM15809v2
Bacillus_sp._CH30_1T	GCF_008364765.1_ASM836476v1
Enterococcus_sp._HMSC073E09	GCF_001813275.1_ASM181327v1
Gulosibacter_chungangensis	GCF_008831155.1_ASM883115v1
Bacillus_sp._VKPM_B-3276	GCF_003990255.1_ASM399025v1
Afipia_sp._OHSU_II-uncloned	GCF_000497555.1_OHSU_uncloned
Alteraurantiacibacter_buctensis	GCF_009827655.1_ASM982765v1
Clostridium_sp._AM34-11AC	GCF_003435375.1_ASM343537v1
Acinetobacter_sp._KU_011TH	GCF_004360215.1_ASM436021v1
Cellulomonas_sp._B6	GCF_001462455.1_ASM146245v1
Citrobacter_sp._RHBSTW-01013	GCF_013781985.1_ASM1378198v1
Bacteroides_bouchesdurhonensis	GCF_902363195.1_MGYG-HGUT-00098
Catellatospora_coxensis	GCF_016862635.1_ASM1686263v1
Bacillus_nealsonii	GCF_000401235.1_BacNeaAAU1

Flavobacterium_undicola	GCF_009909155.1_ASM990915v1
Flavobacterium_turcicum	GCF_011365745.1_ASM1136574v1
Aeromonas_sp._HZM	GCF_000701085.1_AeroHZM_1.0
Cellulomonas_fimi	GCF_013004555.1_ASM1300455v1
Bosea_sp._Tri-39	GCF_004123815.1_ASM412381v1
Acetobacter_sp._DmW_125126	GCF_008690725.1_ASM869072v1
Acinetobacter_soli	GCF_002204165.1_ASM220416v1
Candidatus_Pseudoruminococcus_merdavium	GCF_904419745.1_Chicken_3_mag_1
Aquibacillus_sp._HU2P27	GCF_014280935.1_ASM1428093v1
Dongshaea_marina	GCF_003072645.1_ASM307264v1
Goodfellowiella_sp._AN110305	GCF_008386585.1_ASM838658v1
Bacillus_massiliglaciei	GCF_900098925.1_PRJEB15625
Acinetobacter_sp._YH16051	GCF_013417135.1_ASM1341713v1
Afipia_sp._P52-10	GCF_000516555.1_P52-10
Enterobacter_sp._MGH119	GCF_001037675.1_Ente_cloa_comple
Finegoldia_sp._BIOML-A1	GCF_009678325.1_ASM967832v1
Alicyclobacillus_acidocaldarius	GCF_000219875.1_ASM21987v1
Alteromonas_sp._07-89-2	GCF_013113925.1_ASM1311392v1
Corynebacterium_accolens	GCF_000146485.1_ASM14648v1
Bacillus_sp._HUB-I-004	GCF_004801765.1_ASM480176v1
Exiguobacterium_sp._Leaf196	GCF_001423965.1_Leaf196
Campylobacter_blaseri	GCF_003015205.1_ASM301520v1
Bradyrhizobium_sp._WSM4349	GCF_000373125.1_ASM37312v1
Enterococcus_sp._S77	GCF_016625435.1_S77
Acidovorax_sp._T1	GCF_002176815.1_ASM217681v1
Colwellia_demingiae	GCF_007954275.1_ASM795427v1
Acidovorax_oryzae	GCF_000687165.1_ASM68716v1
Glaciimonas_immobilis	GCF_014202815.1_ASM1420281v1
Desulfatibacillum_alkenivorans	GCF_900142135.1_IMG-taxon_25854
Corynebacterium_sp._319	GCF_008830655.1_ASM883065v1
Dolosigranulum_pigrum	GCF_003264085.1_ASM326408v1
Lachnospira_eligens	GCF_003466985.1_ASM346698v1
Gloeomargarita_lithophora	GCF_001870225.1_ASM187022v1
Halomonas_chromatireducens	GCF_001545155.1_ASM154515v1
Atlantibacter_subterranea	GCF_007570865.1_ASM757086v1
Algoriphagus_aestuariicola	GCF_017254835.1_ASM1725483v1
Faecalimonas_umbilicata	GCF_004346095.1_ASM434609v1
Amycolatopsis_sp._SID8362	GCF_010550475.1_ASM1055047v1
Bacillus_sp._Root131	GCF_001429475.1_Root131
Gilliamella_sp._B14448G7	GCF_016101785.1_ASM1610178v1
Alkalihalobacillus_plakortidis	GCF_001420645.1_ASM142064v1
Bosea_sp._Tri-54	GCF_004123895.1_ASM412389v1
Anaerotruncus_sp._22A2-44	GCF_003665055.1_ASM366505v1

Azospirillum_sp._B506	GCF_000333595.1_ASM33359v1
Flavobacterium_sp._YO12	GCF_004119495.1_ASM411949v1
Curtobacterium_sp._YR515	GCF_900113095.1_IMG-taxon_2616644
Aureimonas_sp._Leaf324	GCF_001423245.1_Leaf324
Alkalibacterium_sp._AK22	GCF_000585255.1_AlkLun1.0
Bartonella_doshiae	GCF_000278155.1_Bart_dosh_NCTC_12
Chiayiivirga_flava	GCF_014202825.1_ASM1420282v1
Aster_yellows_witches_-broom_phytoplasma	GCF_000012225.1_ASM1222v1
Halonatronum_saccharophilum	GCF_000517025.1_ASM51702v1
Acinetobacter_halotolerans	GCF_004208515.1_ASM420851v1
Gordonia_namibiensis	GCF_000298235.1_ASM29823v1
Bacillus_sp._7884-1	GCF_002272245.1_assembly-7884-1_v
Fuerstia_marisgermanicae	GCF_001983935.1_ASM198393v1
Frischella_perrara	GCF_000807275.1_ASM80727v1
Acinetobacter_sp._WCHAc060042	GCF_003231195.1_ASM323119v1
Buttiauxella_sp._JUb87	GCF_004361975.1_ASM436197v1
Bradyrhizobium_sp._LTSP857	GCF_000938285.1_ASM93828v1
Grimontia_sedimenti	GCF_011045095.1_ASM1104509v1
Frankia_sp._EI5c	GCF_001636565.1_ASM163656v1
Bosea_sp._SSUT16	GCF_014764685.1_ASM1476468v1
Kosmotoga_olearia	GCF_000023325.1_ASM2332v1
Dielma_fastidiosa	GCF_003475485.1_ASM347548v1
Dyadobacter_tibetensis	GCF_000566685.1_Dyadobacter_tibet
Candidatus_Babela_massiliensis	GCF_000513475.1_Bbla
Campylobacter_sp._BCW_6872	GCF_001762345.1_ASM176234v1
Cellulomonas_sp._HD19AZ1	GCF_004535805.1_ASM453580v1
Gemmobacter_intermedius	GCF_004054105.1_ASM405410v1
Janthinobacterium_sp._K2E3	GCF_014200725.1_ASM1420072v1
Klebsiella_sp._Kd70_TUC-EEAOC	GCF_002880715.1_ASM288071v1
Georgenia_subflava	GCF_009193155.1_ASM919315v1
Acinetobacter_sp._YH12142	GCF_013418435.1_ASM1341843v1

Record of Contributions

Mike's Version:

I was responsible for building the infrastructure for sourcing/downloading data and testing our compression at the beginning. I then put together the work in Section 1, testing the feasibility of dictionary-based and string-based references for universal reference compression.

Edward worked on the reference compression of similar genomes, creating and implementing a fantastic algorithm that achieved 557x compression. Edward coordinated the direction/vision of the paper.

Antara investigated a range of methods of compressing individual genomes and performing analysis on the compressed data, including implementing the Burrows-Wheeler Transform with run-length encoding. Antara also did a range of analysis on this compressed data, including the Burrows-Wheeler pattern matching algorithm.

Edward's Version:

We decided that, given the short duration of the project, it would be best if everyone did their own thing (all under the same umbrella) than if we tried to collaborate somehow, in which case half our time would likely have been wasted trying to understand each other's code, etc.

As a result, the sections in the paper reflects our individual contributions. While we proof read each other's final writeups and met weekly to discuss our progress and overall direction, we operated very independently.

Mike was responsible for section 1, and investigated ways to compress individual genomes. Not reflected in the paper, in the first week he also created a tool to automate downloading of random genomes to make later exploration easier.

I (Edward) was responsible for creating, implementing and testing the 557x compression algorithm (and previous versions thereof).

Antara worked on creating an implementation of Burrow's Wheeler which would work on compressed data.

I'm happy with what I managed to accomplish – I think it's decently cool that at least on this particular dataset the algorithm managed to beat the state of the art. Working with the others was fun and I think that the project as a whole came together very nicely.

Antara's Version:

Mike created a handy genome downloading and compression benchmarking tool during the first week or so that I used to download datasets and test out compression functions. He was in charge of investigating universal dictionary-based and string-based references, and wrote, and generated the figures for, section 1. He was also the one to get the LaTeX template together for the paper.

Edward coordinated our work; he arranged meetings, established timelines/deadlines, etc. He worked on compression of multiple alignments, and wrote, and generated the figures for, section 2. He also wrote the first drafts of the abstract and introduction, which guided how we organized the paper overall.

I worked on implementing compression using the Burrows-Wheeler Transform with run-length encoding, as well as the Burrows-Wheeler pattern matching algorithm using the compressed data. I wrote, and generated the figures for, section 3.