A GENOME COMPRESSION ALGORITHM SUPPORTING MANIPULATION

Lenwood S. Heath, Ao-ping Hou, Huadong Xia, and Liqing Zhang*
Department of Computer Science, Virginia Tech,
Blacksburg, VA 24061, USA
Email: {heath,aphou,xhd,lqzhang}@vt.edu

With the advent of the thousand dollar genome, one can anticipate the need to store, communicate, and manipulate many human genomes. Data compression methods have been developed to store and communicate genomes efficiently. Unfortunately, these methods do not support efficient manipulation (e.g., subsequence retrieval) of the compressed genome. We develop a data compression scheme that achieves both efficient storage and efficient sequence manipulation. We demonstrate the practicality of the method on two databases of genomes, one for the human mitochondrion and one for the H3N2 virus. In both cases, we achieve high compression ratios and $O(\log n)$ subsequence retrieval times.

1. INTRODUCTION

The human genome has about 3 billion DNA base pairs (BPs), consisting of 23 chromosomes with lengths ranging from about 33 to 247 million BPs. If a researcher or physician is dealing with many human genomes, then there is a challenge to store, communicate, and manipulate those genomes. Data compression techniques can address the storage and communication challenges. A manipulation task might involve retrieving a number of random single nucleotide polymorphisms (SNPs). Unless a data compression technique is carefully designed, such manipulation will require the full decompression of the genome before the SNPs can be retrieved.

Our study defines an efficient data structure for storing genomic sequences and a fast algorithm to randomly access subsequences. We achieve a 98.8% compression ratio when compressing 5473 mitochondrial sequences and a 97.3% compression ratio for H3N2 virus genomes. Furthermore, the algorithm allows users to quickly compute statistics on various types of mutations and to retrieve complete or partial genomic sequences.

The contents of this paper are as follows. Section 2 describes other genome compression studies. Section 3 introduces our data structure and algorithm. Section 4 gives results for two sets of genomic sequences. Section 5 is discussion, and Section 6 concludes.

2. RELATED WORK

Compression of sequential data has been of interest for decades[11]. More recently, compression of genomic data has become a focus. Most genome compression algorithms in the literature aim to compress each single genomic sequence directly (GenCompress[3], DNACompress[4], GSCompress[8], compression using the Maximum Likelihood Model[9]). Those algorithms apply either statistical methods or dictionary based schemes to compress genomic data directly[8]. Although they exploit properties of genomic sequential data, such as palindromes or approximate repeats, to the best of our knowledge, the best results ever reported are by GSCompress[8] with compression ratios of 87-91%. The compression ratios of other algorithms are about 78%[4, 7]. Most of the algorithms mentioned try to capture the limited variation patterns occurring in the sequences. Therefore, it is better to compress similar sequences together. The results presented in those papers are the output from compressing a number of similar sequences. One reason that GSCompress achieved the best results in terms of compression ratios might be because they compress all sequences of an organism together, exploiting the high similarity among sequences.

The idea of compressing human genomes by encoding their differences with a reference genome is a relatively new idea presented by two groups from UCI[2, 5]. The motivation for encoding only the difference comes from the fact that about 99.9% of any two

*Corresponding author.

human genomes are identical to each other[10]. Therefore, a delta (difference) representation that encodes the differences between two human genomes can be quite small[5]. Although a reference sequence is required to retrieve the information from delta representations, a higher compression ratio is achieved by amortizing the cost over many genomes. As reported in Brandon et al.[2], a 433-fold level of compression can be achieved with an appropriate reference sequence for the data set of 3615 mitochondria genomic sequences, which is significantly better than previous work that compresses single genomic sequences.

Our work is also a delta compression. However, our algorithm differs in that we focus not only on compression but also on efficient manipulation. We apply our algorithm to compress 5473 mitochondria genomes from Genbank and achieve a compression ratio of 98.8%. Our results represent an intentional tradeoff between compression efficiency and manipulation efficiency. In our scheme, absolute position values can be reached without traversing all previous variations, which is required of previous methods.

Although the compression ratio of our algorithm is a bit lower than the UCI groups, our result is much better than other cited work.

3. METHODS

3.1. Overview

Our genome compression strategy can be divided into four stages, depicted in Figure 1. First, a set (or database) of genomic sequences, called target genomes, are preprocessed for classification and alignment. Second, we extract the differences between each target genome and a fixed reference genome. Using those differences, we can retrieve partial and complete sequences with the corresponding reference genome. Third, to achieve more efficient compression, we further compress the differences with Huffman codes. Finally, we use those differences to extract mutation statistics in each target genome and to support retrieval of genomic sequences.

3.2. Data Preprocessing

The first stage is to preprocess a database of genomic sequences to classify them into several sets. It also accomplishes pairwise sequence alignment of each target genome with the reference genome. For example, a human has 23 chromosomes, and each chromosome is aligned with the corresponding chromosome of the reference genome. For H3N2 viruses, the genome consists of 11 segments, which we treat as 11 chromosomes. After grouping sequences based on chromosomes or segments, we use the multiple sequence alignment application MUSCLE[6] to align the sequences with the reference genome.

3.3. Difference Extraction

To describe differences as instructions consisting of an operator, a position, and BPs, we define 5 kinds of operators occurring between reference and target genomes. The operators are insertion, deletion, replacement, insertion after replacement, and deletion after replacement. Figure 2 shows examples of the instructions. All of the genome positions in this paper refer to the absolute positions on the reference genomes. The first example, in Figure 2a, is inserting 2 BPs (G and A) at position 140 of the reference genome. The second example, in Figure 2b, is deleting 2 BPs at position 151. The third example, in Figure 2c, replaces 3 BPs (CTT to GAA) at position 161. Generally, the three operators (insertion, deletion, and replacement) are sufficient to represent the differences in two genomic sequences and to develop the difference set of a target genome.

However, using only three basic operators causes overlapping positions in the difference set. For instance, the differences in Figure 2d are replace (170, A) and insert (170, GA). This is ambiguous if we simply put the two encoded differences (instructions) together: we do not know whether the content at position 170 is A or G. To remove the ambiguity, we have to define an order between the two instructions. That means we have to use at least one additional bit to encode the difference. In addition to that, we have to read through and find out all instructions related to a specific position in order to decompress the content of that position. This increases not only implementation complexity, but also algorithmic complexity. Therefore, we define two additional instructions, "insertion after replacement" and "deletion after replacement" to make the search unique. Figure 2d shows combined replacement and

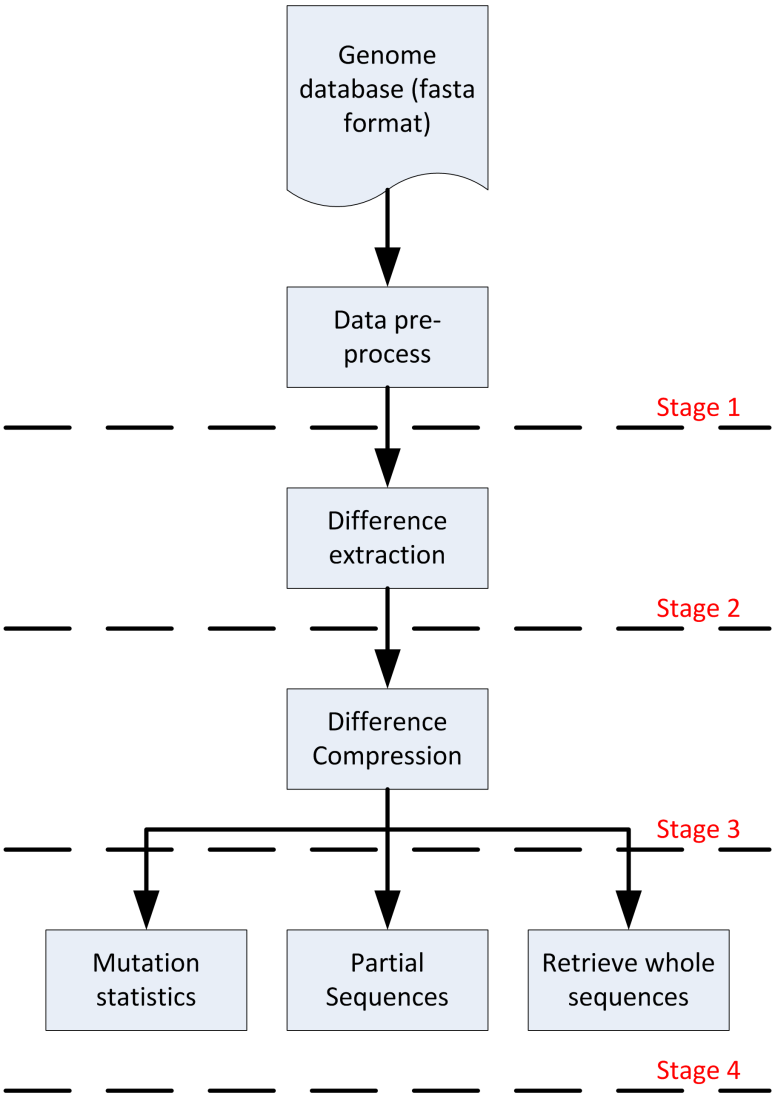

Fig. 1. Genome Compression Flow. Stage 1. Categorize and retrieve sequences for preprocessing. Stage 2. Extract the differences between the reference genome and each target genome. Stage 3. Encode the differences using hybrid Huffman coding. Stage 4. Provide statistics and support efficient retrieval for researchers.

insertion. Figure 2e gives another example combining replacement and deletion.

3.4. Difference Compression

Once we extract the differences from the target genomes, we encode the instructions with Huffman codes. According to Figure 2, an instruction consists of three different fields: operator, position, and BPs. Because the maximum size of a human chromosome does not exceed 250 million BPs, the stored position utilizes only the last 29 bits out of a 32-bit integer type. Hence, it is efficient to combine an operator with 3 bits and a position with 29 bits into a new 32-bit integer type in our method. Likewise, the combined value is also defined as position earlier. After the combination, we extract the Huffman codes for position and BPs.

Figure 3 shows the flow of encoding instructions with a Huffman code. First, we count the frequencies of the different items appearing in the complete genome for position and BPs. Next, we construct two mapping tables for positions and base pairs to give each item an index. Then, we build Huffman codes for each table according to the frequencies. Finally, we translate each difference into the corresponding

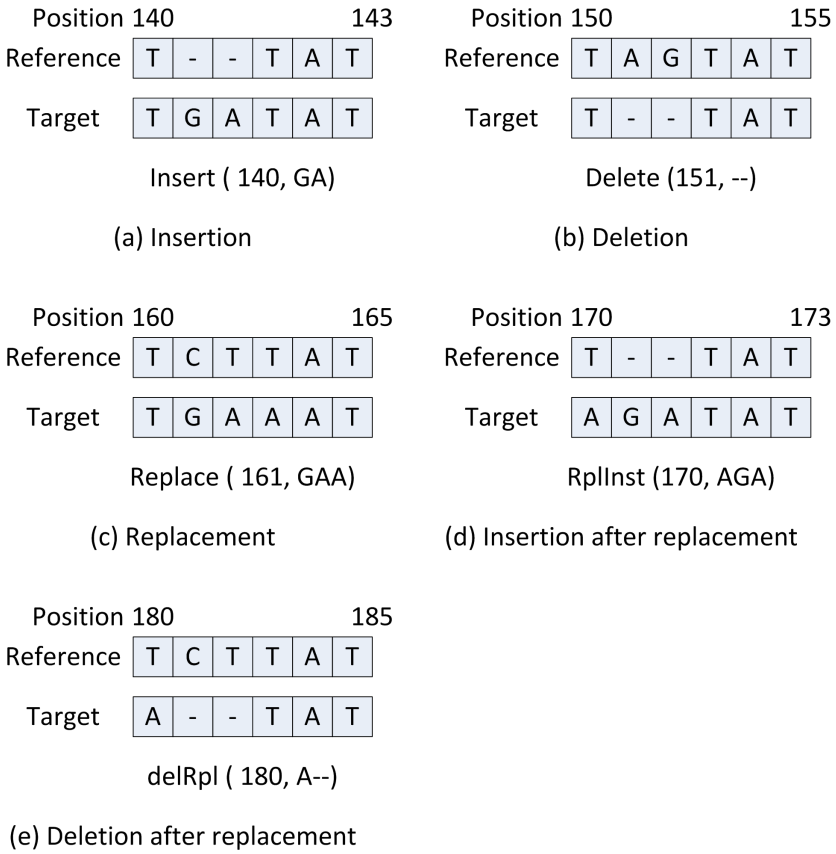

Fig. 2. The operators describing 5 kinds of differences. Transforming those conditions to five instructions with reference genome's position makes it unique in a target genome. (a) The example of insertion is that the target genome has 2 extra BPs at position 140 and 141. (b) The example of deletion is that the target genome misses 2 extra BPs at positions 151 and 152. (c) The example of replacement is that the target genome has 3 different BPs at positions 161 to 163. (d) The example of insertion after replacement is that the target genome replaces only 1 base pair and inserts 2 extra BPs at positions 170 to 172. (e) The example of deletion after replacement is that the target genome has only 1 different BP and misses 2 extra BPs at positions 180 to 182.

Huffman code.

Actually, the values encoded or decoded with a Huffman code are related to the mapping table indices rather than positions or BPs. For example, to encode the instruction, *replace (161, GAA)*, the compression process combines the operator (replace; code=3) and position (161) into the value 0x600000A1. After counting the frequencies of the value and GAA respectively, the value falls in the 14th entry of the position mapping table, and string GAA falls in the 12th entry of the base pair mapping table. By constructing a Huffman tree for each of these two tables, the Huffman codes of 14 and 12 are 10001 and 10011 respectively. Finally, we write 0x600000A1 into the 14th entry of the position mapping table and 10001 into the 14th entry of the position Huffman table. Besides, we write "GAA" into the 12th entry of the base-pair mapping table and 10011 into the 12th entry of the base-pair Huffman table.

After constructing the tables, we encode the differences for each genome. During the encoding of the instructions, we use binary search to seek the corresponding indices of a position and BPs. Then, we index the Huffman tables to seek the codes. Finally, we store them onto disk. However, if there is no corresponding item in the tables, we store the primitive value binding with position and operators or a character string for BPs. Storing primitive data and compressed data in the same file makes our compression more flexible to permit a new target genome to use the old tables for compression.

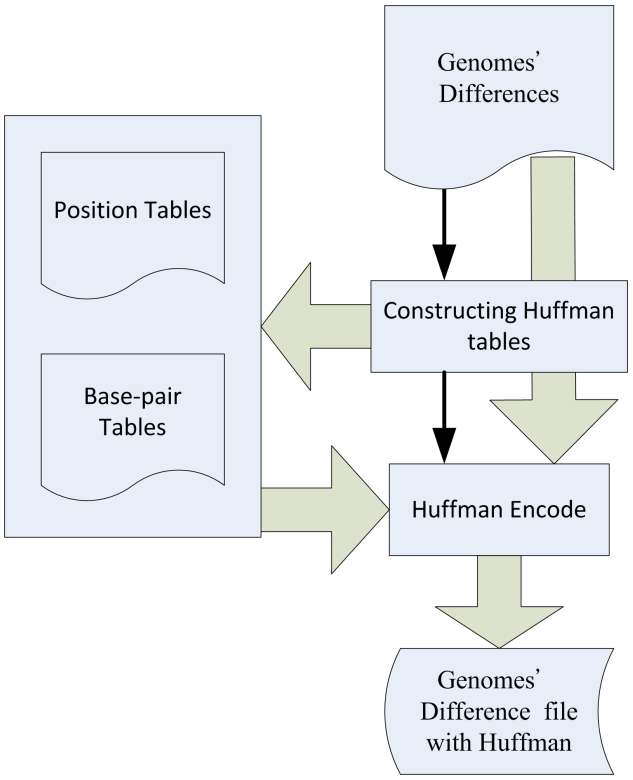

Fig. 3. The compression flow for Huffman coding. After extracted differences, we construct Huffman tables by collecting statistics of positions and BPs among different target genomes in order to store frequent position or BPs with short code length. Once we have the Huffman codes for position and BPs, we encode all difference files for further compression with Huffman codes.

3.5. Data Structure for Extension of New Genomes

Storing the compressed instructions and primitive instructions in the same file requires an efficient data structure. Figure 4 is the data structure used in our compression method. In this structure, the first bit of each byte is defined as the beginning of an instruction. Using this bit can assist searching to reach an instruction in a specific position without having to decode from the beginning. Furthermore, we can do a binary search with this bit identifying an instruction from any disk position. Next, the remaining 7 bits store the difference consisting of a position and BPs. Due to hybrid compression, there are two bits to identify whether the compressed data has a Huffman code or not. This is because some Huffman codes of the rare positions or BPs are longer than non-coded positions or BPs. Hence, we need to use a hybrid data structure to optimize compression size. Since we encode positions and BPs with/without Huffman codes, there are 2 bits used for position and BPs to identify Huffman/non-Huffman codes. Finally, we store the compressed or primitive difference after those control bits.

3.6. Compression Applications

There are three applications (manipulations) developed on our compressed database: getting statistics of mutations, retrieving partial target genomic sequences, and recovering the entire target genome. Because our instructions include the absolute position of the reference genome, one can easily retrieve BPs at any position or a range of positions. Figure 5 illustrates our search engine. First, we get the last instruction before the requested position from the input and jump to the position in the reference genome. Next, we access the reference genome and target differences sequentially, because the request from input is usually to retrieve a portion of a genomic sequence. For example, when retrieving 200 BPs from position 1234, the method fetches the last instruction before 1234. Then, it reconstructs the following 200

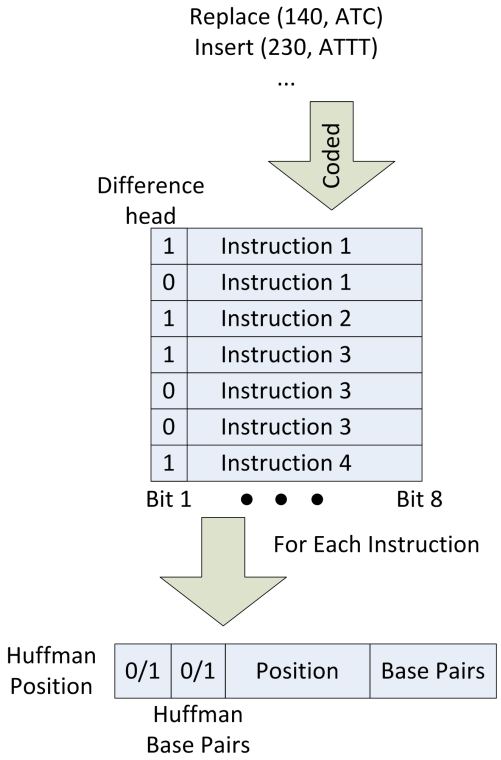

Fig. 4. The compression data structure. To support adding a new target genome to the current database, the data structure records Huffman coding method or non-Huffman method for positions and BPs. Once the decoder attains the coding information, it decodes positions and BPs either by the Huffman decoder or as primitive data.

BPs. Therefore, the most time-consuming stage of the search is the binary search of the last instruction before the requested position; its time complexity is $O(\log n)$ where n is the number of instructions for compressing a sequence.

4. RESULTS

In this section, we present our results with H3N2 genomic sequences and human mitochondrial genomic sequences.

4.1. Genome Sequences

We use two sets of genomic sequences, H3N2 viral genomic sequences and human mitochondrial genomic sequences. The first is retrieved from the virus set (`http://www.fludb.org`), and the second is retrieved from GenBank. Because there are at most 11 segments in an H3N2 genome, we arbitrarily select 1 reference segment for each segment from the H3N2 genomic sequences. For the mitochondrial reference genome, we use the revised Cambridge Reference Sequence (GenBank accession number: NC_012920). Altogether, there are 1455 H3N2 genomic sequences and 5473 mitochondrial sequences compressed with our compression algorithm. The average length of 11 H3N2 segment sequences is less than 1400 BPs. The average length of the mitochondrial sequences is 16,569 BPs. Our computational platform consists of an Intel T770 2GHz processor with 3GB memory, and the operating system is Linux Fedora 8.

4.2. Compression Results

To compare with other compression algorithms, we use gzip and bzip2 to compress primitive genomic sequences. Table 1 shows the sizes of mitochondrial sequences and H3N2 sequences with different compression methods. The first two columns contain the results of general compression tools. The third column contains the results of encoding text-based sequences to binary codes with position/base-pair tables. The last column contains the results using Huffman codes for compression. Comparing mitochon-

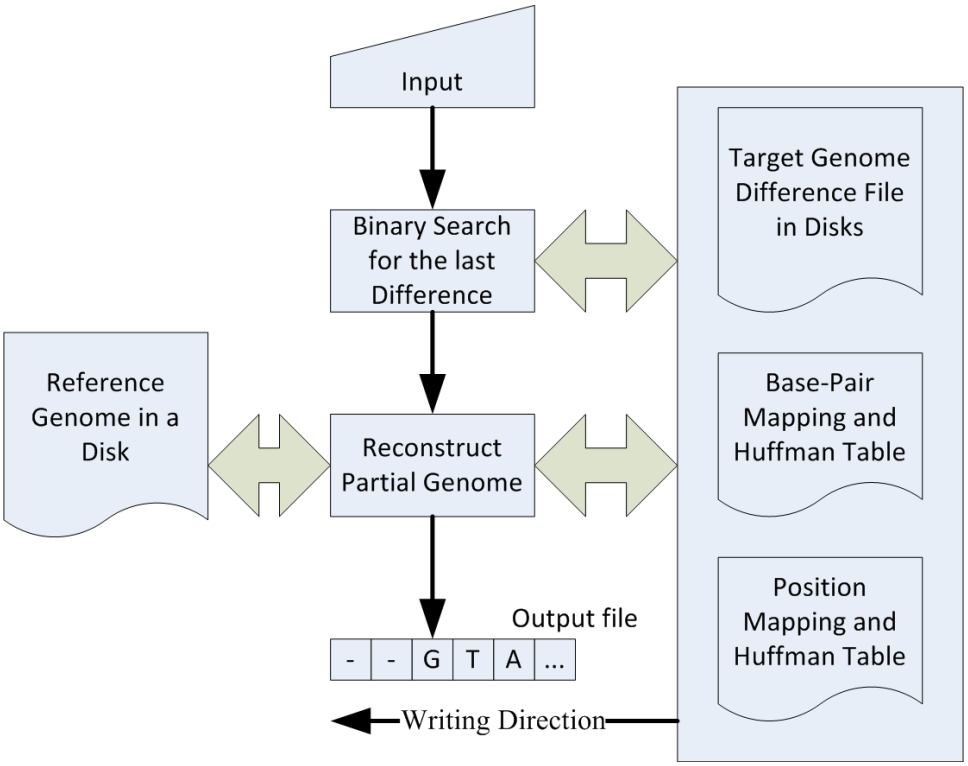

Fig. 5. Once commands are received, the search engine uses binary search for the last instruction before the required position of the command. After reaching the instruction, the method retrieves parts of the reference genome and instructions in order to retrieve the primitive target genomic sequence.

Table 1. Comparison with other tools

	gzip	bzip2	DNACompress	GenCompress	binary	Huffman
Mitochondrion (Primitive size 91,590,508 bytes)						
Size	10,860,887	3,096,944	80,271[a]	46,213[a]	2,211,120	1,101,628
Ratio	0.8815	0.9662	0.7612	0.8617*	0.9759	0.9880
H3N2 (Primitive size 1,995,960 bytes)						
Size	101,494	54,314	474,139	25,837	73,901	52,927
Ratio	0.9492	0.9728	0.7626	0.9870	0.9630	0.9735

[a] Since GenCompress cannot deal with large data sets and DNACompress compresses only single sequence files, only 20 randomly selected mitochondrial sequences are compressed for comparison. Therefore, the file size shown here is not of the same order as found in other columns.

drial sequences and H3N2 genomic sequences, the mitochondrial sequences achieve more efficient compression than H3N2 sequences. This might be due to the fact that there are more mitochondrial sequences than H3N2 sequences and mitochondrial sequences are longer than H3N2 sequences; more Huffman code can be shared by each mitochondrial genomic sequence as a result. Thus, increasing the number of genomic sequences and the sequence length can improve our compression efficiency.

Both gzip and bzip2 are general compression tools. To compare our algorithm to those specific to genomic compression, we pick two algorithms, GenCompress and DNACompress[3, 4], for which we can find the authors' implementations, and run those programs with the same data sets we used. The results are also listed in Table 1. All the results are achieved by compressing all sequences in a batch, except for the result indicated with *, when GenCompress compresses the entire mitochondrial data set. Because GenCompress crashes while compressing 20 mitochondrial sequences together, the result with * is achieved by compressing mitochondrial sequences one by one. The performance of DNACom-

press is not only worse than ours but also worse than general compression software. GenCompress has the best compression ratio for the H3N2 data set, but ours are still comparable to it. GenCompress performs poorly on the mitochondrial data set, partly because the software from the authors has to process the mitochondrial sequences one by one. But we conjecture that it is not a good idea to build one lengthy table containing all variation patterns for the whole 5473 mitochondrial data set even with a faultless implementation. So it is likely that our algorithm will eventually beat such direct compression software for large data sets, such as the mitochondrial one.

Table 2 shows the execution time of our algorithm for compressing the H3N2 and mitochondrial databases. The first row is the time to make alignments and extract differences. The second row is the time to construct Huffman tables. The third row is the time to convert the original genomic database to a compressed database. Finally, the fourth row is the time to retrieve the original database from the compressed database. Although the total compression time for the mitochondrial sequences is over 24 hours, our algorithm can finished the compression and decompression in seconds or minutes. This is because most of the compression time is spent on alignment. Hence, our algorithm reaches the requirement of $O(\log n)$ complexity in decoding parts.

Table 2. Time of Encoding and Decoding H3N2 and mitochondrial sequences

	H3N2	Mitochondrial
Alignment/ Difference (Difference)	103.84s (3.47s)	23.56Hr (159.14s)
Huffman table	2.38s	17.57s
Convert	13.59s	936.333s
Retrieve	11.02s	54.17s

4.3. Application Showcase

In the above, we have shown how our algorithm facilitates the storage and transmission of genomes by efficient compression. Another benefit of our algorithm, and the more important aspect, is that our compressed data maintains a superior structure via flexible indexing, which enables efficient manipulation and analysis of the data. Here, we showcase how our methods help to explore or manipulate genomes facilitated by such flexible indexing.

It is easy with our software to analyze distribution patterns of differences among genomes, as we pre-index every difference in the delta representation. Figure 6 shows an example. It gives the distribution map of all differences (termed mutations) among the whole population of 5473 mitochondrial genomes. The delta map reveals interesting features of the genomic variation. For example, it is straightforward to find in the figure that mutations of the first 500 positions and last 500 positions have a much higher probability to occur than those of the positions in between. This helps to discover a pattern that otherwise might be hidden: at the head and tail of the mitochondrial genomic sequences, mutations are very likely to happen. Actually, further analysis verifies this observation: 840 out of 5473 mitochondria sequences have a missing head and/or tail compared to the reference genome, although they are claimed to be "complete" in the Genbank database. This example exemplifies how the positions and mutations our software organizes reveal common variation patterns in the population of the genomes.

5. DISCUSSION

In previous sections, we have demonstrated how we use Huffman coding to compress the delta information that we generated. The reason that Huffman coding is effective here is due to the characteristics of our target data sets. The genomic sequences, whether viral, mitochondrial, or full human genomes, are special data. First, two genomes from two individuals have high similarity to each other. Second, the differences among many genomes can also be classified into a limited set of patterns with highly skewed distributions. By analyzing and retrieving such patterns from a database of genomes, we can design more efficient delta representations for specific populations of genomes.

Here we compute some statistics for the differences of the 5473 complete mitochondria sequences from GenBank. Figure 7 shows the frequency distribution of all the instructions. In the figure, each point on the x-axis represents a unique instruction that is ordered according to its number of occurrences in the data set. Similarly, Figure 8 shows

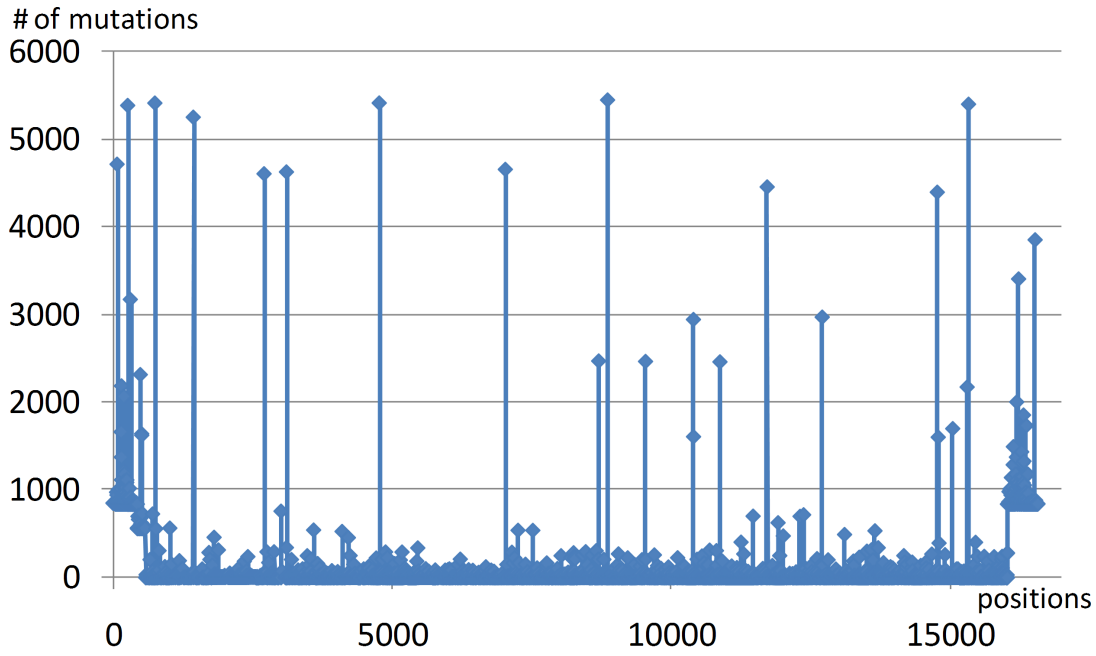

Fig. 6. The mutation distribution.

the frequency distribution of positions where instructions happen. We see that both figures show a highly skewed distribution. A small set of items (instructions or positions) have a very high frequency of appearance in difference sets, while a multitude of other items occur only a few times in the genomic data set. Such skewed data distributions suggest that we could apply a code system like Huffman coding to compress those items with respect to their distribution. Experiments in Section 4 also demonstrate the possibility to apply mixed codes of Huffman coding and direct bit compression.

Another interesting observation is that the two distributions are almost the same, not only qualitatively, but also quantitatively. If we define operation as (operator + BPs) and deem instruction as (position+operation), then this implies that positions are highly correlated with operations. To verify this observation, we compare the number of mutated positions to that of instructions. For the 5473 mitochondrial sequences, there are a total of 5436 positions that have some instruction; and there are 6391 instructions in total occurring in these positions. This means that only 1.176 instructions are encoded with one position on average. Therefore, possibly there is no need to encode the position separately from the operations. To combine the two into one item and compute Huffman codes for the whole instruction naturally leads to space saving during compression.

Such statistical analysis is also conducted over the set of 1455 H3N2 virus sequences. Their distributions (Figure 9 and Figure 10) show very similar difference patterns as those of mitochondrial sequences. We conjecture that the patterns that we observed are common to most genomes that we aim to compress and manipulate. Such generality is easily explainable biologically. Therefore, we believe the presented compression code applies to most other genomic data as well.

6. CONCLUSIONS AND FUTURE WORK

This paper proposes a framework to compress and manipulate large groups of genomic sequences. We compare a target genome with a selected reference genome and represent the target genome using the identified differences. Such differences not only reduce the space to store a genome, they also preserve the alignment information in terms of the reference

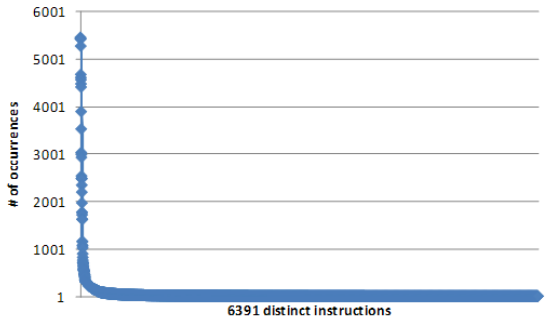

Fig. 7. Frequency Distribution of Different Instructions (Mitochondria data set)

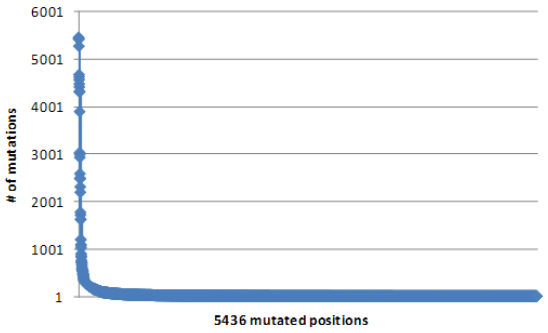

Fig. 8. Frequency Distribution of Changed Positions (Mitochondria data set)

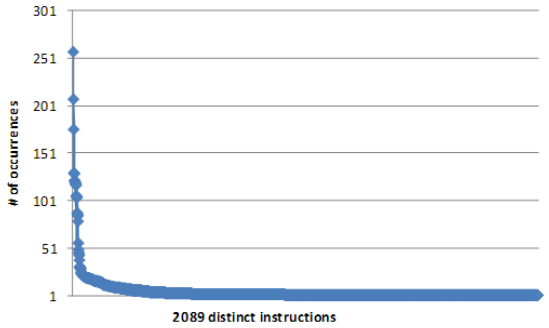

Fig. 9. Frequency Distribution of Different Instructions (H3N2 data set)

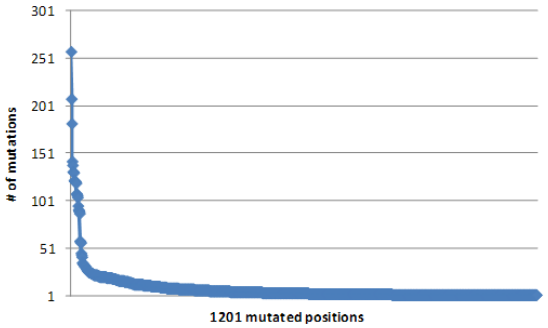

Fig. 10. Frequency Distribution of Changed Positions (H3N2 data set)

genome. Therefore, by carefully designing the data structure to store those differences, we create an efficient indexing scheme that facilitates fast search for and access to compressed genomes. The experimental results demonstrate that our solution enables the flexibility of manipulating genomic sequences, while achieving a high compression ratio.

The presented work is a prototype to exemplify the framework we propose. There are still research topics to explore to either improve the manipulation efficiency or achieve better compression ratios. One thing is that we only apply a bit-level representation and a simple usage of Huffman coding to compress the delta information. It is expected that we could further increase the compression ratio by utilizing mature techniques of direct genome compression that do not destroy our indexing structure.

Another issue is the choice of the reference genome for the algorithm. In the paper, we choose the Revised Cambridge Reference Sequence (rCRS) to compress the mitochondrial data set, because the rCRS was firstly presented in 1981[1] and has been used as a standard reference sequence extensively thereafter. It has become the *de facto* sequence when people extract SNPs for their mitochondrial sequences. Using the rCRS as the reference genome, we can conveniently produce SNPs consistent with others. However, for compression purposes, it is better to extract the central point of the whole data set and use it as the reference genome. In this way, we can achieve the best performance in term of the compression ratio. And such a reference genome can be easily synthesized by choosing the value of the majority nucleotide at each position.

In this paper, we apply binary search as the indexing scheme to access/search genomic sequences. A better solution is to design a perfect hash function to index the delta representation. Hashing is a classic data structure in computer science for rapid access to data that is identified by a key. Most hashing research assumes dynamic data, so that data items can be inserted or deleted at any time. In our ap-

plication, we can assume that data are static, since genomes do not change. We will design indexing schemes based on such perfect hash functions and compare the performance to binary search to evaluate the time/space tradeoff.

ACKNOWLEDGMENTS

The work was supported by NSF grant IIS-0710945 to L.Z.

References

1. S. Anderson, A. T. Bankier, B. G. Barrell, M. H. de Bruijn, A. R. Coulson, J. Drouin, I. C. Eperon, D. P. Nierlich, B. A. Roe, F. Sanger, P. H. Schreier, A. J. Smith, R. Staden, and I. G. Young, *Sequence and organization of the human mitochondrial genome*, Nature, 290 (1981), pp. 457–465.
2. M. C. Brandon, D. C. Wallace, and P. Baldi, *Data structures and compression algorithms for genomic sequence data*, Bioinformatics, 25 (2009), pp. 1731–1738.
3. X. Chen, S. Kwong, and M. Li, *A compression algorithm for DNA sequences and its applications in genome comparison*, in The Tenth Workshop on Genome Informatics (GIW'99), 1999, pp. 52–61.
4. X. Chen, M. Li, B. Ma, and J. Tromp, *DNACompress: Fast and effective DNA sequence compression*, Bioinformatics, 18 (2002), pp. 1696–1698.
5. S. Christley, Y. Lu, C. Li, and X. Xie, *Human genomes as email attachments*, Bioinformatics, 25 (2009), pp. 274–275.
6. R. C. Edgar, *MUSCLE: Multiple sequence alignment with high accuracy and high throughput*, Nucleic Acids Research, 32 (2004), pp. 1792–1797.
7. T. Matsumoto, K. Sadakane, and H. Imai, *Biological sequence compression algorithms*, Genome Informatics, 11 (2000), pp. 43–52.
8. H. Sato, T. Yoshioka, A. Konagaya, and T. Toyoda, *DNA data compression in the post genome era*, Genome Informatics, 12 (2001), pp. 512–514.
9. I. Tabus and G. Korodi, *Genome compression using normalized maximum likelihood models for constrained Markov sources*, in Information Theory Workshop, ITW '08. IEEE, 2008, pp. 261–265.
10. J. C. Venter et al., *The sequence of the human genome*, Science, 291 (2001), pp. 1304–1351.
11. J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory, 23 (1977), pp. 337–343.