LFQC: A lossless compression algorithm for FASTQ files
Sudipta Pathak1, Sanguthevar Rajasekaran1*
1Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269-4155

ABSTRACT
Motivation: Next Generation Sequencing (NGS) technologies have revolutionized genomic research by reducing the cost of whole genome sequencing. One of the biggest challenges posed by modern sequencing technology is economic storage of NGS data. Storing raw data is infeasible because of its enormous size and high redundancy. In this paper we address the problem of storage and transmission of large fastq files using innovative compression techniques.

Results: We introduce a new lossless non-reference based fastq compression algorithm named LFQC. We have compared our algorithm with other state of the art big data compression algorithms namely gzip, bzip2, fastqz (Bonfield and Mahoney, 2013), fqzcomp (Bonfield and Mahoney, 2013), G-SQZ (Tembe, et al., 2010), SCALCE (Hach, et al., 2012), Quip (Jones, et al., 2012), DSRC (Deorowicz, et al., 2011), DSRC-LZ (Deorowicz, et al., 2011), etc. This comparison reveals that our algorithm achieves better compression ratios. The improvement obtained is up to 225%. For example, on one of the data sets (SRR065390), the average improvement (over all the algorithms compared) is 74.62%.

Availability: The implementations are freely available for non-commercial purposes. They can be downloaded from http://engr.uconn.edu/~rajasek/FastqPrograms.zip.

Contact: rajasek@engr.uconn.edu

1 INTRODUCTION
With the advancement of Next Generation Sequencing (NGS), researchers have started facing the problem of storing vast amounts of data. Petabytes of data produced by modern sequencing technologies demand huge infrastructure to store them on disks for current or future analysis. The rate of increase of NGS data volume has quickly outpaced the rate of decrease in hardware cost and as a consequence scientists have realized the need for new sophisticated software for economic storage of raw data. Moreover, transmission of gigantic files produced by sequencing techniques will cost huge amounts of time causing a significant delay in research and analysis. One other fact observed by the scientists is that the NGS data files contain huge redundancies and could be efficiently compressed before transmission. Numerous general purpose compression algorithms can be found in the literature. However, these algorithms have been shown to perform poorly on sequence data. Hence, time and energy were invested to develop novel domain specific algorithms for compression of big biological data files. As revealed in (Giancarlo and Scaturro 2009), compression of FASTQ files is an important area of research in computational biology. The NGS data are generally stored in FASTQ format. FASTQ files consist of millions to billions of records and each record has the following four lines:

- Line 1 stores an identifier. It begins with @ followed by a sequence identifier and an optional description.
- Line 2 represents the read sequence. This is essentially a sequence of letters.
- Line 3 begins with the + character which is sometimes followed by the same sequence identifier as line 1.
- Line 4 represents the Quality Scores. This is a string of characters. Line 2 and line 4 must be of equal lengths.

Compression of nucleotide sequences has been an interesting problem for a long time. (Cox, et al., 2012) apply Burrows-Wheeler Transform (BWT) for compression of genomic sequences. GReEn (Pinho, et al., 2012) is a reference based sequence compression method offering a very good compression ratio. (Compression ratio refers to the ratio of the original data size to the compressed data size.) Compressing the sequences along with quality scores and identifiers is a very different problem. Tembe, et al. (Tembe, et al., 2010) have attached the Q-scores with the corresponding DNA bases to generate new symbols out of the base-Q-score pair. They encoded each such distinct symbol using Huffman encoding (Huffman 1952). (Deorowicz and Grabowski 2011) divided the quality scores into three sorts of quality streams:

1. quasi-random with mild dependence on quality score positions.
2. quasi-random quality scores with strings ending with several # characters.
3. quality scores with strong local correlations within individual records.

To represent case 2 they use a bit flag. Any string with that specific bit flag is processed to remove all trailing #. They divide the...
quality scores into individual blocks and apply Huffman coding. (Tembe, et al., 2010) and (Deorowicz and Grabowski 2011) also show that general purpose compression algorithms do not perform well and domain specific algorithms are indeed required for efficient compression of FASTQ files. In their papers they demonstrate that significant improvements in compression ratio can be achieved using domain specific algorithms compared to bzip2 and gzip. The literature on FASTQ compression can be divided into two categories, namely, lossless and lossy. A lossless compression scheme is one where we preserve the entire data in the compressed file. On the contrary, lossy compression techniques allow some of the less important components of data to be lost during compression. (Wan, et al., 2011) propose a lossy quality score encoding scheme based on a binning technique. (Kozanitis, et al., 2010) perform randomized rounding to perform lossy encoding. (Asnani, et al., 2012) introduce a lossy compression technique for quality score encoding. (Wan, et al., 2012) proposed both lossy and lossless transformations for sequence compression and encoding. When it comes to medical data compression it is very difficult to identify which components are unimportant. Hence, many researchers believe that lossless compression techniques are particularly needed for biological/medical data. Quip (Jones, et al., 2012) is one such lossless compression tool. It separately compresses the identifier, sequence and quality scores. Quip makes use of Markov Chains for encoding sequences and quality scores. G-SQZ (Tembe, et al., 2010) and DRSC & DSRC-LZ (Deorowicz, et al., 2011) are also considered state of the art lossless compression algorithms which we compare with ours in this paper. (Bonfield and Mahoney 2013) have come up with a set of algorithms to compress FASTQ files recently. They perform identifier compression by storing the difference between the current identifier and the previous identifier. Sequence compression is performed by using a set of techniques including base pair compaction, encoding, and an order-k model. Apart from hashing, they have also used a technique for encoding quality values by prediction. (Kozanitis, et al., 2010) and (Fritz, et al., 2011) have contributed a reference based compression technique. The idea is to align the reads using a reference genome. This approach stores the genomic locations and any mismatches instead of storing the sequences. A major advantage of reference based compression is that as the read length increases it produces much better compression ratios compared to non-reference based algorithms. In spite of providing excellent sequence compression ratios, reference based compression techniques have many disadvantages. The success of these algorithms solely depends on availability of a good reference genome database which may not always be ensured. Creation of such databases itself is a challenging problem. Also reference based compression techniques are not self contained. Decompression requires the exact reference genome to match the genomic locations and extract the reads. Hence, it is important to preserve the reference genome.

In this paper we present a lossless non-reference based FASTQ compression algorithm called LFQC (Lossless FastQ Compressor) that can elegantly run on commodity machines. The algorithm is provisioned to run in in-core as well as out-of-core settings. The rest of the paper is organized as follows: Section 2 describes the FASTQ structure and provides details on steps involved in compressing FASTQ files. Section 3 supplies details on decompression algorithms. In Section 4 we compare LFQC with other state of the art algorithms and provide performance results. In Sections 5 and 6 we provide future directions and conclusions, respectively.

## 2 FASTQ FILES COMPRESSION

A FASTQ file consists of records where each record has four lines. An example record is shown below.

```
@SRR013051.81530PTX:0:14:518
AGTTGATCCACCTGAGAATTAGGA
I11F10H1E1+515I1HBC(1B)
```

Several FASTQ compression algorithms found in the literature compress the four fields of the records separately. We also employ this approach in our algorithm LFQC.

### 2.1 Compression of the identifier field

The identifier field is generated very systematically. This property can be utilized to compress this field. We provide an example of Illumina generated identifier below.

```
HWUSI - EAS68761DAJ:8:1:1055:3384/1
```

The identifier consists of the following set of fields: instrument name, flow-cell lane, tile-number within flow-cell lane, ‘x’-coordinate of the cluster within the tile, ‘y’-coordinate of the cluster within the tile, index number for multiplexed sample and a field indicating whether or not the sequence is a member of a pair. Most of the time the identifier is preceded by the data set name. After studying a considerable number of FASTQ files we found that the identifier can be divided into four types of tokens.

- **Type 1**: Tokens having data that do not change from one record to the next.
- **Type 2**: Tokens having the same data value over a set of consecutive records.
- **Type 3**: Tokens having integer values that are monotonically incremented or decremented over consecutive records.
- **Type 4**: Tokens having data not belonging to any of the above mentioned types.

The base caller algorithm produces the DNA sequences and a probability value $P$ for each base in the sequence. This value $P$ is the probability of the base being incorrectly called. These error probabilities are then quantized to integers. In the algorithm of (Ewing and Green 1998), the error probabilities are transformed into PHRED quality scores and stored. The transformation is given by

$$Q = -10 \times \log_{10} P$$

Then, the quality scores $Q$s are truncated and fitted in the range of 0 to 93. Each integer is then incremented by 33 so that the values range from 33 to 126. This is done to be able to print the scores in the form of printable ASCII characters. This format is known as SANGER-format (Cock, et al., 2010).
Our algorithm for compressing the identifiers works as follows. First, we partition each identifier into tokens. Let $I$ be any identifier. Let $I_{j,i}$ represent the substring of $I$ from the $i$th position through the $j$th position. We scan through each identifier $I$ and extract $I_{j,i}$ as a token if any of the following occurs:

- $I_{j,i}$ and $I_{j+1}$ belong to any of the following set $C$ of characters: dot(,), space( ), underscore( _ ), hyphen(-), slash(/) or colon(:). We call these symbols as Pivot Symbols.
- $I_{j,i}$ is a digit and $I_{j}$ is a non-digit or vice versa. $I_{j-1}$ is a digit and $I_{j}$ is a non-digit or vice versa.

The following example provides an explanation of how an identifier is divided into tokens. Let the identifier be @SRR007215.1135HWUSI − EAS867.61DAJ : 8 : 1 : 1055 : 3384/1. We split the identifier into the following tokens:

- Token 1 : @SRR
- Token 2 : 007215
- Token 3 : 1135
- Token 4 : HWUSI
- Token 5 : EAS
- Token 6 : 867
- Token 7 : 61
- Token 8 : DAJ
- Token 9 : 8
- Token 10 : 1
- Token 11 : 1055
- Token 12 : 3384
- Token 13 : 1

This procedure of splitting the identifier is called tokenization. Tokenization also produces a regular expression formed of place holders, essentially the token numbers and the character set $C$ mentioned above. This regular expression helps us to rebuild the identifier file during decompression. Regular expression for the above example is the following: $T_1T_2\cdot T_3T_4 - T_5T_6\cdot T_7T_8 : T_9 : T_{10} : T_{11} : T_{12}/T_{13}$, where $T_i$ represents the $i$th token. Decompression algorithm replaces each $T_i$ with the respective value and builds the identifier.

Pseudocode for The Tokenization Algorithm is given in Algorithm 1.

Let the number of tokens present in each identifier be $t$. Let the tokens present in identifier $I^g$ be $T_1^g, T_2^g, T_3^g, \ldots , T_t^g$. Let the identifiers from all the input records be $I^1, I^2, \ldots , I^N$. Let $T_i$ stand for the set of all $i$th tokens from all the identifiers, for $1 \leq i \leq t$. Each $T_i$ is called a token set ($1 \leq i \leq t$). We first construct these token sets. Followed by this, we have to choose an appropriate technique to compress each of the token sets. We let $\tau_i$ denote the compressed $T_i$, for $1 \leq i \leq t$. The compression techniques we have employed are listed below.

- RunLength Encoding: Token sets with the same token values over a set of consecutive records are compressed using RunLength Encoding. For token sets with constant values throughout the data set also, we use the same compression technique.
- Incremental Encoding: Token sets having integer values monotonically incremented or decremented over consecutive records are compressed using Incremental Encoding.
- Token sets with tokens not belonging to any of the above categories are not compressed.

As discussed above the Identifier Compression Algorithm also generates a regular expression used for reconstruction of the original text after decompression of individual compressed token set texts. We provide the pseudocode for Identifier Compression Algorithm and Incremental Encoding in Algorithm 2 and Algorithm 3, respectively. The Identifier Compression Algorithm also records the encoding algorithms used for the compression of each $T_i$, for $1 \leq i \leq t$. Identifier Compression Algorithm stores the compressed token sets $\tau_i$ in individual files. The regular expression given by RegEx is stored in a separate file too.

### 2.2 Quality Score Compression

In this section we consider the problem of compressing quality scores. There will be a quality score for every read. The length of a quality sequence will be the same as that of the read. There is a quality score for every character in the read. Let $Q = q_1, q_2, q_3, \ldots , q_n$ be any quality score sequence. A careful study of quality sequences reveals that there exists a strong correlation between any quality score $q_i$ and the preceding some number of quality scores. The correlation gradually decreases as we move away from $q_i$. The rate of change of correlation varies from one file to the other almost randomly. This nature of FASTQ quality scores makes it challenging to efficiently encode for compression. We have also observed that many sequences end with quality scores 2 and this is a known issue with Illumina base caller. Taking all this into account we have decided to encode quality scores using Huffman Encoding. Applying Huffman Encoding on the entire set of quality scores does not produce a good compression ratio. We have tried to come up with a reversible transformation which would split the quality score text into several parts. We would then separately encode each part using Huffman Encoding. We first split the sequences into non-overlapping $l$-mers with an empirically decided $l$-value. In general, it ranges between 4 and 12. For each $l$-mer we count the frequency of distinct quality scores in it. Assume that the quality score $q_j$ has a frequency of $f_j$ in $l$-mer $L_j$. We pick the quality score $q_j$ with the largest frequency $f_j$ in $L_j$ of $L_j$. $L_j$ goes to the $q_j^{th}$ bucket. $l$-mers where none of the symbols showed majority of occurrences go to a special bucket called generic bucket $B_G$. We also maintain an index file maintaining records of which $l$-mer is mapped to which bucket. This file is also compressed using Huffman Encoding. This index file incurs a space overhead on the compression technique. The pseudocode for Quality Score Compression is provided in Algorithm 4.

The quality score compression technique is improved using two more simple techniques. First, we get rid of all trailing #. For many of the FASTQ files we observe that after # no quality score occurs. In these cases, removing # improves the compression ratio to a good extent. Next, each of the buckets $B_{q_i}$ is checked to see if run length encoding offers a better compression ratio than Huffman coding. It
is easy to find out whether or not to go for run length encoding just by checking the average run length of each quality score sequence.

### 2.3 Sequence Compression

We have used the same algorithm as the quality scores for sequence compression. The FASTQ sequences are strings of 5 possible characters namely A, C, G, T, N. We first remove all the 'N's from the sequence text because the 'N's always get a quality value of 0. Hence these could be easily retrieved from decompressed quality score text. Next we apply the same algorithm as mentioned above in Quality Score Compression.

### 3 DECOMPRESSION

Decompression algorithms are inverse to the way compressions are performed. We use two distinct algorithms for decomposition, one for identifier decompression and the other for sequence and quality score decompression.

#### 3.1 Identifier Decompression

In decompression each compressed text $T_i$ is decoded using the corresponding algorithm. Details are in Algorithm 5. \textit{RegEx} provides the reconstruction formula for each identifier. Each place holder $t_i$ in the \textit{RegEx} is replaced with the corresponding decompressed token set $T_i$. We provide the Incremental Decoding Algorithm in Algorithm 6.

#### 3.2 Quality Score and Sequence Decompression Algorithm

As we make use of the same algorithm for compression of sequences and quality scores we combine them into a single algorithm. The decompression of quality scores is just the reverse and straightforward. We first decode the Huffman encoded texts given by $\tau_{B_i}$ and $\tau_{B_{index}} \forall i$. Let the decoded texts be $B_i$ and $B_{index}$. The entry in $B_{index}$ denoted by $B_{index}$, says from which bucket to collect the next $l$-mer. These $l$-mers are then placed sequentially next to each other to get the original text $Q_T$. Algorithm 7 provides the algorithm for decompression of quality scores. In the case of sequence decompression we add the 'N's at the end.

### 4 EXPERIMENTAL RESULTS

We have compared the performance of our algorithm LFQC with other state of the art algorithms and report the findings in this section. We have compared our algorithm with general purpose compression algorithms like gzip and bzip2 and also a set of algorithms specific to the domain namely fastqz, fqzcomp, SCALCE, Quip, G-SQZ, DSRC and DSRC-LZ. It is important to note that our algorithm does not accept any input other than a raw FASTQ file. Hence, we do not bring into comparison algorithms that accept any reference genome, performs any sorting or alignment using a reference genome. Some of the algorithms assume that the FASTQ files are stored in unicode format where each character takes 2 bytes on disk. We calculate all the compression ratios based on the assumption that each character on disk occupies 1 byte. In other words, we work on the raw FASTQ files without any change and without taking any extra information from outside. Ours is a lossless compression algorithm and performs an overall compression of the entire FASTQ file. To perform a fair comparison we have ignored all lossy compression techniques and algorithms compressing only the sequence or quality score fields.

The data sets used in our experiments are all publicly available. We downloaded them from 1000 Genome Project or data repositories pointed to by the references that we cite. Our aim is to develop an elegant compression algorithm that can run on commodity machines. To achieve this goal we have developed our software to support out-of-core implementation. All the experiments are performed on a hardware with the following specifications: 2.6 GHz Intel processor and 6 GB of primary memory. It is important to mention that the software can be configured to run in-core by changing just a single parameter. For a fair comparison we performed all the experiments with in-core settings. Sometimes we had to sample the large data sets to fit into main memory. Note that, this has no impact on the compression ratio reported.

Table 1 shows a comparison of performances of LFQC relative to some of the state of the art algorithms. We chose SRR027520_1 and SRR065390_1 because we found most of the algorithms are using these data sets for evaluating their performance. Our algorithm runs in only one mode targeting to achieve the best compression ratio in real time. The best results are boldfaced in the table. As far as compression ratios are concerned we are the clear winner. The key to success of our compression algorithm is that we perform extremely well when it comes to identifier and quality score compression. The sequence compression we obtain is comparable to what other non-reference based algorithms achieve. Splitting the identifier enables us to apply run length encoding on the tokens. We capture the parts that remain unchanged over the entire set of identifiers and maintain only a single copy with a count. This saves us a huge amount of space. We call this type of compression vertical compression. Tokens where we could not apply vertical compression like run length or incremental encoding, are encoded using Huffman encoding. These fields are generally the ‘Y’-coordinates which are strings of digits. We get a decent compression ratio there too. Apart from identifiers, we get excellent compression ratios in quality scores too. Our idea of splitting the quality score text into $l$-mers pays back by mapping the $l$-mers with a dominating quality score (quality score with more than 50% occurrence) to its corresponding bucket. Huffman coding or run length encoding on individual buckets ensures that the dominating quality score always gets a single bit. Although we need to store an index file for mapping the $l$-mers back to decompressed file, the overhead due to this is negligible. Only for data sets with almost random quality scores we apply straight forward Huffman encoding. In terms of compression rate and decompression rate fastqz(fast) and gzip are the winners, respectively. As the table shows fastqz(fast) achieves the best compression rate at the cost of compression ratio. Both gzip and fastqz(fast) offer pretty low compression ratios compared to other algorithms. We achieve the highest and the second highest improvement in compression ratios against fastqz(fast) and gzip, respectively. The experimental results support the trade off between compression ratio and compression/decompression speed as mentioned in (Bonfield and Mahoney 2013).

Table 2 and Table 3 compare the performance of a set of algorithms with ours on benchmark data sets. We divided the
<table>
<thead>
<tr>
<th>Program</th>
<th>SRR027520_1</th>
<th></th>
<th>SRR065390_1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ratio</td>
<td>% Improvement</td>
<td>C.R.</td>
<td>D.R.</td>
</tr>
<tr>
<td>gzip</td>
<td>2.85</td>
<td>96.49</td>
<td>12.2</td>
<td>3.56</td>
</tr>
<tr>
<td>bzip2</td>
<td>3.44</td>
<td>62.79</td>
<td>7.0</td>
<td>13.0</td>
</tr>
<tr>
<td>SCALCE(fast)</td>
<td>3.72</td>
<td>51.35</td>
<td>9.4</td>
<td>25.4</td>
</tr>
<tr>
<td>SCALCE(slow)</td>
<td>4.01</td>
<td>38.95</td>
<td>7.8</td>
<td>13.1</td>
</tr>
<tr>
<td>DSRC(fast)</td>
<td>4.03</td>
<td>38.95</td>
<td>24.7</td>
<td>32.9</td>
</tr>
<tr>
<td>DSRC(slow)</td>
<td>4.03</td>
<td>38.95</td>
<td>13.5</td>
<td>32.2</td>
</tr>
<tr>
<td>Quiq(fast)</td>
<td>4.46</td>
<td>23.06</td>
<td>16.8</td>
<td>13.7</td>
</tr>
<tr>
<td>Quiq(slow)</td>
<td>4.51</td>
<td>23.06</td>
<td>8.3</td>
<td>13.7</td>
</tr>
<tr>
<td>fastqz(fast)</td>
<td>2.57</td>
<td>117.89</td>
<td>36.1</td>
<td>32.8</td>
</tr>
<tr>
<td>fastqz(slow)</td>
<td>4.55</td>
<td>23.07</td>
<td>4.6</td>
<td>3.8</td>
</tr>
<tr>
<td>fqzcomp(fast)</td>
<td>4.45</td>
<td>25.84</td>
<td>31.4</td>
<td>32.5</td>
</tr>
<tr>
<td>fqzcomp(medium)</td>
<td>4.55</td>
<td>23.07</td>
<td>22.0</td>
<td>13.7</td>
</tr>
<tr>
<td>fqzcomp(slow)</td>
<td>4.60</td>
<td>21.73</td>
<td>8.2</td>
<td>8.3</td>
</tr>
<tr>
<td>SeqSqueezeI(slow)</td>
<td>4.57</td>
<td>22.53</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>LFQC</td>
<td>5.60</td>
<td>NA</td>
<td>4.5</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Table 1. Performance comparison. Ratio is defined as the ratio of the original file size to the compressed file size, % Improvement measures the improvement in compression ratio of LFQC over other algorithms. Specifically, this % is calculated as \((\text{LFQC compression ratio} - \text{the other algorithm’s compression ratio}) \times 100\) / \text{the other algorithm’s compression ratio}\). C.R. and D.R. are the compression and decompression rates in MBs/Sec.

<table>
<thead>
<tr>
<th>Program</th>
<th>SRR013951_2</th>
<th></th>
<th>SRR027520_2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ratio</td>
<td>% Improvement</td>
<td>C.R.</td>
<td>D.R.</td>
</tr>
<tr>
<td>gzip</td>
<td>2.41</td>
<td>82.15</td>
<td>3.9</td>
<td>81.0</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.81</td>
<td>56.22</td>
<td>6.4</td>
<td>53.3</td>
</tr>
<tr>
<td>GSQZ</td>
<td>2.77</td>
<td>58.48</td>
<td>2.7</td>
<td>193</td>
</tr>
<tr>
<td>DSRC</td>
<td>3.23</td>
<td>35.91</td>
<td>29.1</td>
<td>3.90</td>
</tr>
<tr>
<td>DSRC(LZ)</td>
<td>3.29</td>
<td>33.43</td>
<td>28.3</td>
<td>3.97</td>
</tr>
<tr>
<td>LFQC</td>
<td>4.39</td>
<td>NA</td>
<td>4.3</td>
<td>4.51</td>
</tr>
</tbody>
</table>

Table 2. Performance comparison. Ratio is defined as the original file size upon compressed file size, % Improvement measures the improvement in compression ratio of LFQC over other algorithms. Specifically, this % improvement is calculated as \((\text{LFQC compression ratio} - \text{the other algorithm’s compression ratio}) \times 100\) / \text{the other algorithm’s compression ratio}\). C.R. and D.R. are the compression and decompression rates in MBs/Sec.

Comparisons into tables according to the experiments and results reported in the literature. For SRR007215_1 and SRR010637 we see marginal improvements in the compression ratio. The reason is that our quality compression algorithm performs poorly for these data sets. The distribution of quality scores in these data sets is close to random. Hence, the bucket allocation technique dumps almost 70% of the quality scores in the generic bucket producing no significant improvement. The identifier and sequence compression algorithms continue to produce better results though. In terms of compression ratio our algorithm consistently performs better while the gzip offers the best compression/decompression speed.

This idea works in practice due to the reason that the neighboring quality scores are highly correlated. Due to slow execution rate of Huffman coding program our compression and decompression rate is not competitive enough. This is an area we are working on to improve. Apart from that, we would like to add an option to receive a reference genome to our algorithm to improve the sequence compression ratio. Our algorithm is totally based on Huffman Encoding, a prefix coding technique, for its performance. We are also pursuing other non-prefix coding techniques for compression to overcome any bottleneck Huffman Encoding suffers from.

**Funding:** This research has been supported in part by the following grants: NIH grant R01-LM010101 and NSF 1447711.

**5 DISCUSSION**

Our idea of binning the \(l\)−mers of quality scores helps us to achieve a better compression ratio than any of the state of art algorithms. This ensures that in each bucket there is abundance of one particular quality score. This quality score is then encoded using just a single bit by Huffman Encoding. Even though we have to maintain an index for decompression, the overall compression ratio is high.

6 REFERENCES

<table>
<thead>
<tr>
<th>Program</th>
<th>SRR007215_1</th>
<th></th>
<th></th>
<th>SRR010637</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>4.19</td>
<td>56.08</td>
<td>9.1</td>
<td>92.7</td>
<td>52.01</td>
<td>7.6</td>
</tr>
<tr>
<td>bzip2</td>
<td>5.21</td>
<td>25.52</td>
<td>4.2</td>
<td>23.9</td>
<td>4.25</td>
<td>24.47</td>
</tr>
<tr>
<td>GSQZ</td>
<td>4.71</td>
<td>38.85</td>
<td>25.2</td>
<td>4.8</td>
<td>3.98</td>
<td>32.91</td>
</tr>
<tr>
<td>DSRC</td>
<td>6.51</td>
<td>0.46</td>
<td>21.0</td>
<td>35.8</td>
<td>5.13</td>
<td>3.11</td>
</tr>
<tr>
<td>DSRCLZ</td>
<td>6.51</td>
<td>0.46</td>
<td>21.0</td>
<td>35.8</td>
<td>5.13</td>
<td>3.11</td>
</tr>
<tr>
<td>LFQC</td>
<td>6.54</td>
<td>NA</td>
<td>4.1</td>
<td>4.2</td>
<td>5.29</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 3. Performance comparison. Ratio is defined as the original file size upon compressed file size; % Improvement measures the improvement in compression ratio of LFQC over other algorithms. Specifically, this % improvement is calculated as (LFQC compression ratio - the other algorithm’s compression ratio)*100/(the other algorithm’s compression ratio). C.R. and D.R. are the compression and decompression rates in MBs/Sec.


Algorithm 1

1: procedure **TOKENIZATION**
2: \[ C \leftarrow \{., \ldots, -, \ldots, \} \]
3: \[ D \text{ is the set of all digits} \]
4: \[ A \text{ is the set of all characters} \]
5: \[ \text{for each identifier } I \text{ in the input do} \]
6: \[ \text{for } 1 \leq i < j \leq n \text{ (where } n = |I|) \text{ do} \]
7: \[ \text{if } (I_{i-1} \in C \text{ and } I_{j+1} \in C) \text{ or } ((I_{i-1} \in D \text{ and } I_{j+1} \notin D) \text{ or } (I_{i-1} \notin D \text{ and } I_{j+1} \notin A)) \]
8: \[ T_k \leftarrow I_{i,j} \text{ where } T_k \text{ is the } k^{th} \text{ token and } I_{i,j} \text{ represents the substring of } I \text{ from} \]
9: \[ \text{position } i \text{ through position } j. \]
10: \[ \text{end if} \]
11: \[ \text{end for} \]
12: \[ \text{end for} \]
Algorithm 2

1: **procedure** IDENTIFIER COMPRESSION
2: Form the token sets using the algorithm TOKENIZATION.
3: Let RegEx be an empty regular expression.
4: **for** each token set $T_i$ **do**
5: Compress $T_i$ using an appropriate encoding algorithm as mentioned above. Let the encoded text be $\tau_i$.
6: Append $\tau_i$ to RegEx as a place holder for token set $T_i$.
7: Append $C_j$ to RegEx where $C_j$ is the set of pivot symbols for $T_i$.
8: **end for**

Algorithm 3

1: **procedure** INCREMENTAL ENCODING
2: **for** $1 \leq i \leq t$ **do**
3: Let the tokens in $T_i$ be $T_{i1}, T_{i2}, \ldots , T_{iN}$.
4: if $T_{i1}, T_{i2}, \ldots , T_{iN}$ is a monotonically increasing or a monotonically decreasing sequence where the difference between two successive elements is 1
5: $\tau_i = T_{i1} - 1$ where $\tau_i$ is the compressed text.
6: **end if**
7: **end for**

Algorithm 4

1: **procedure** QUALITY SCORE COMPRESSION
2: $Q_T \leftarrow$ Quality Score Text
3: $B_{q_i}$ = Bucket allocated for $l$–mers with more than 50% $q_i \forall i$.
4: **for** each $l$–mer $u \in Q_T$ **do**
5: if $\exists i$ such that $f_{q_i} \geq \frac{1}{4}$
6: (where $f_{q_i}$ is frequency of $q_i$ in $u$) **do**
7: send $u$ to $B_{q_i}$; Also record the fact that $u$ is in $B_{q_i}$
8: in a data structure $B_{index}$.
9: **else**
10: send $u$ to $BG$.
11: **end if**
12: **end for**
13: Encode all the buckets $B_{q_i}$ and $B_{index}$ using Huffman Encoding. Let the encoded texts be $\tau_{B_{q_i}}$ and $\tau_{B_{index}}$.

Algorithm 5

1: **procedure** IDENTIFIER DECOMPRESSSION
2: **for** each compressed text $\tau_i$ **do**
3: Apply the appropriate decoding algorithm to get the decompressed text $T_i$.
4: **end for**
5: Replace each place holder $t_i$ in RegEx with the corresponding decompressed text $T_i$.

Algorithm 6

1: **procedure** INCREMENTAL DECODING
2: Parse $\tau_i$. Let, $V_{start}$ = starting value.
3: $T_1 = V_{start}$
4: $T_j+1 = T_j + 1$ for $j = 1, 2, 3, \ldots , N - 1$ where $N$ is the total number of records.

Algorithm 7

1: **procedure** QUALITY SCORE DECOMPRESSSION
2: Apply Huffman Decoding on $\tau_{B_{q_i}}$ and $\tau_{B_{index}}$ to decode the texts. Let the decoded texts be $B_{q_i}$ and $B_{index}$, respectively.
3: $Q_T =$ decoded quality score text to be formed.
4: **for** each entry $k \in B_{index}$ **do**
5: Go to bucket id given by $k$ and get the next $l$–mer.
6: Append this $l$–mer to the end of $Q_T$.
7: **end for**