ISSN:0975-9646

A Compression Algorithm for Nucleotide Data Based on Differential Direct Coding and Variable Length Look up Table (LUT)

Govind Prasad Arya
Scholar,Uttarakhand Technical University, Dehradun
&
Computer Science & Engg Department
Shivalik College of Engineering, Dehradun

R. K. Bharti
Computer Science & Engg Department
Bipin Tripathi Kumaon Institute of Technology, Dwarahat

Abstract-The ongoing exponential increase of genomic data, together with full diploid human genomes, creates new challenges not only for understanding genomic structure, function and development, but also for the storage, navigation and privacy of genomic data. In this paper, we have proposed a modified Direct Differential Coding algorithm. It is a general purposed nucleotide compression algorithm based on variable length LUT. Here the method identifies repeat regions in the individual sequence and the repeat regions are store in the look-up table (LUT). This algorithm compresses both repeat and non repeat sequences. It also handles the non base character and compresses any nucleotide sequences. It gives better result as compared to existing algorithm.

The Differential Direct Coding algorithm was a fixed size look-up table algorithm i.e. it used a table of fixed size containing the 64 maximum possible combinations of the triplets obtained by combination of four characters A, G, T and C. We make this table of variable length by adding some more combinations in the look-up table, which are of the size of multiple of triplet i.e. their size is (6,9.12….) since the number of ACSII characters available were not utilized completely. Our algorithm is based on longest common substitution (LCS). It searches a longest common sequence in multiple of 3 and then substitutes an ASCII value in the place of that sequence to generate variable length LUT. In the previous algorithms, the compression ratio so obtained was smaller as compared to the variable length LUT compression algorithm which creates a relatively massive difference when the algorithm is applied on the large genomic repositories. In addition to this, our algorithm also utilizes the maximum number of ASCII characters which are available, thus increasing the efficiency.

BACKGROUND

The area dealing with the storage of the biological data of living organisms, forces us to use the database management system to store the data. The basic need is to warehouse this data, which carries the sequences of large sizes, lying in the databases.

Genbank is one of the biggest databases for biological sequences, whose size roughly doubles in every 18 months [1]. Though in present scenario, space availability is not considered as a big problem, since the high capacity storage devices are easily available in low costs, still the compression of these biological data is of great concern due to many factors like fast searching and retrieval of the data and for performing operations on them.

There are many methods to achieve the compression of the data. For e.g. pointer method, table method, etc. [3, 4, 5, 6, 7]. In this research we will particularly concentrate on the table method. There exist many algorithms based on dictionary method like Ziv-Lampel. There exist some other arithmetic encoding algorithms also like Huffman algorithm. However, these universal text compression algorithms are not suitable for compression of biological sequences as they consider the sequence as a pure text stream. If we talk about the DNA sequences, we know that it deals with only four symbols representing four nucleotide bases {A, C, T, G}. these four symbols could have been modeled as {00, 01, 10, 11} respectively, where we can observe that every nucleotide base occupying 8 bit is made to occupy 2 bits, when encoded in the above mentioned binary form. This could have been one of the most efficient encoding schemes, if and only if there were no other symbols in the sequence, other than A, G, T and C base characters. Here, though the encoding can be done, but main problem will occur during decompression as the binary code of the unexpected symbol like N or S which will definitely match with the binary code of A, G, T and C. An another type of algorithm used for DNA compression is Differential Direct (2D) Coding Algorithm [2], which can overcome this problem by differentiating between the base characters and the unexpected symbols. The 2D coding algorithm uses the group of three characters (triplets), being replaced by some other character.

Shortcomings of 2D Algorithm

a) As discussed, the algorithm stores all the possible combinations of {A, G, T, C, U}, though some of the combinations are not acceptable and are never used, therefore leaving some of the non-printable ASCII characters unused.

b)It only concentrates over the triplets, even though other combinations may yield better compression ratio.

PROPOSED ALGORITHM: DIFFERENTIAL DIRECT CODING WITH VARIABLE LENGTH LOOK-UP TABLE

Whenever we need to compress some biological sequence, we are aware that at a time only a DNA or mRNA sequence will be compressed. Hence, if the case of formation of triplets is considered, with combination of four symbols {A, C, G, T} or {A, C, G, U} for DNA or mRNA respectively, we will encounter maximum 64 combinations. These 64 triplet combinations can be handled by 64 non-printable ASCII characters, whereas there exist total 127 non-printable ASCII characters. Therefore, the remaining 63 characters can be used to store some other combinations of size more than 3, which can yield a better compression. We divide the look-up table into two parts here: fixed length LUT and variable length LUT. The former one will always exist there containing 64 combinations of triplets, but the variable length LUT can be of variable size, containing the combination bases of the size which is in multiple of 3. This will yield to the proper utilization of the available non-printable ASCII characters.

Model: We consider the ASCII characters [8] between the ranges 0 to 127 for the Auxiliary symbols. The other range between -1 to -127 is divided in order to adjust fixed size LUT and variable size LUT. The remaining -128th character is used for encoding the unknown character. The model is shown in Table 1.

Coding: Here we will apply the model described above, for the encoding of input sequence. As the input sequence will begin scanning, we will always search for the triplet from the fixed size LUT. It will further scan next triplet. Therefore the total scanned characters will be of length 6 now. If this combination of 6 characters is available in the variable part lookup table, we will scan next triplet, otherwise we will write the ASCII character corresponding to the last match group in the output file and also insert the current combination into the variable part of look-up table. This process will be carried on till the whole sequence is encoded. But we will always keep in mind that, once the variable size look-up table is filled, we store no more patterns and just search for the combinations from both parts of the table. Also this will be followed by writing the output corresponding to that combination to the output file. This output file will be the required compressed file, when the whole sequence is scanned. While scanning the input sequence, the case may occur where a triplet cannot be formed. Here we will write that single character or a group of two into the output file as it is. Whenever an unexpected symbol is obtained, we will stop scanning the triplets and write that unexpected character's ASCII equivalent, followed by its positive integral repeats in its decimal form. This can be explained in Table 2.

Algorithm

```
Initialize: String s=NULL
Initialize: st=NULL
Initialize: t=NULL
Step 1: read first three unprocessed characters (t). If t!=
NULL, go to step 2.
Else:  process the last one or two characters by step 5.
Step 2: Check that t has all non-N characters. If it is true, go
to step 3.
Else if t has N characters, go to step 4
Else go to step 5.
Step 3: If st exists in the LUT* then s=st Else
 {
 Output the code (character) for s
// signed Byte code (character) that are mapped in LUT
(From -1 to -127).
Add st to the LUT table;
                s=t;
               }
        End if
Step 4: Search first N and successive Ns in the string and
count total number of appearing in successive Ns, replace all
such Ns with "/n**/" into destination file. After this, go to
step 6.
 If number of successive Ns appears more than one time
repeat the step 4.
Step 5: write non-N characters whose number is less than
three into destination file directly without any modification.
After that, go to step 6.
Step 6: Return to step 1 and repeat all process until EOF is
reached.
```

The detailed java programming code is mentioned in APENDIX.

RESULTS

Thus proposed algorithm has high compression ratio to other existing DNA Sequence Compression algorithms. This algorithm also uses less amount of memory as compared to the other algorithms and is easy to implement.

The proposed algorithm compresses Nucleotide sequences like DNA as well as RNA. All other algorithms use only the other properties of sequences such as repeated and non-repeated. If the sequence is compressed using proposed algorithm it will be easier to make sequence analysis between compressed sequences. It will also be easier to make multi sequence alignment. High compression ratio also suggests a highly repetitive sequence. The compression results for differential direct coding using fixed length LUT & variable length LUT are shown in Table 3.

The Existing Algorithm Vs Proposed Algorithm (A Comparative Study)

With the help of fixed length LUT we can compress the DNA sequence up to 1/3 of its original size. But now we can achieve higher compression by using variable length LUT. In this research paper we are making some modification in Differential Direct Codling's Look up Table. Our result shows that the proposed algorithm is better than the existing one. Our algorithm is also based on longest common substitution (LCS), besides substitution of triplets only. The proposed algorithm can provide much better compression as the longer sequences are found frequently in big sequences.

Table1. The 2D Coding With Variable Length LUT Data Model

Type of Data	Description	Range	Look-Up Table
Auxiliary Symbol	ASCII	0 to 127	
Triplet	Set of three Base characters	-1 to -64	Fix LUT
Multiple of Triplet	Set of **6, 9, 12…** Base Characters	-65 to -127	Variable Length LUT
Unknown	?	-128	

Table 2.The 2D Encoding Process with Variable Length LUT

Step	Input Sequence	Triplet(t)	Multiple of Triplet (st)	Look-Up Table		Encoded Sequence (s)
				Status of st	Entry	
1	ACTGCTACTGCTNNNTC					
2	ACT	ACT	ACT	Found	ACT = # GCT = +	
3	GCT	GCT	ACTGCT	Not Found	Add with ACTGCT=$	#
4	ACT	ACT	GCTACT	Not Found	Add with GCTACT=@	#+
5	GCT	GCT	ACTGCT	Found	ACTGCT=$	#+
6	NNN					#+$N3
7	TC	TC	TC	<3 Char		**#+$N3TC**

Table 3.Results & Conclusion

S.N	Type of Sequence	Original size of sequence before compression	Size of Sequence after Compression	
			Using Direct Differential Coding(2D)	Using Direct Differential Variable length LUT
1	atatsgs	9647	3217	3101
2	atef1a23	6022	2008	1957
3	atrdnaf	10014	3338	3276
4	atrdnai	5287	1763	1734
5	chmpxx	15180	5060	4874
6	chntxx	155844	51948	50540
7	hehcmvcg	229354	76452	74736
8	humdystrop	105265	35089	34347
9	humhdabcd	58864	19622	19201
10	vaccg	47912	15972	15374
	Average	**64338.9**	**21446.9**	**20914**

References

1. Benson,D.A. & Karsch-Mizrachi,I., GenBank. Nucleic Acids Res., (2008) 36, D25–D30.

2. Gregory Vey et al., Differential direct coding: a compression algorithm for nucleotide sequence data, Database, (2009), doi: 10.1093/database/bap013.

3. J. Ziv & A. Lempel., A universal algorithm for sequential data compression, (1977) *IEEE Transactions on Information Theory*, vol. IT-23.

4. Behzadi,B. & LeFessant,F., DNA compression challenge revisited. Symposium on Combinatorial Pattern Matching (CPM), (2005), Jeju Island, Korea, Springer, Berlin/Heidelberg,pp. 190–200.

5. Cherniavski,N. & Lander,R., Grammar-based compressionof DNA sequences, (2004) , Computer Science & Engineering Technical Report. University of Washington, 2007-05-02, pp. 1–21.

6. X. Chen & M. Lip, Dnacompress:fast and effective dna sequence compression, (2002), *Bioinformatics*, vol. 18.

7. Bao,S. et al., A DNA sequence compression algorithm based on LUT and LZ77, (2005)

8. Ascii code. [Online]. Avalable: http://www.LookupTables.com

9. Ateet Mehta, (2010), et al., " DNA Compression using Hash Based Data Structure", IJIT&KM, Vol2 No.2, pp. 383-386.

10. Cao,M.D. et al. (2007) A simple statistical algorithm for biological sequence compression. In Proceedings of the IEEE Data Compression Conference (DCC). IEEE Computer Society, pp. 43-52.

11. Galperin,M.Y. and Cochrane,G.R. (2009) Nucleic Acids Research annual Database Issue and the NAR online Molecular Biology Database Collection in 2009. Nucleic Acids Res., 37, D1–D4.

12. Liu,Q., Yang,Y., Chen,C. et al. (2008) RNACompress: grammar-based compression and informational complexity measurement of RNA secondary structure. BMC Bioinformatics, 9, 176.

APENDIX

Java Programming Code

Compress File Button Code for 2D Coding With Variable Length LUT

```
private void
jButton7ActionPerformed(java.awt.event.ActionEvent evt)
{
   String tf;
   long size;
   float f,g=1;

   tf=textField2.getText()+"/"+textField3.getText();
   File fs=new File(textField1.getText());
   File ft=new File(tf);

   size=fs.length();
   textField6.setText(String.valueOf(size)+" Bytes");
   f=100.00f/size;
   try
   {
   RandomAccessFile fr = new RandomAccessFile(fs,"r");
   RandomAccessFile fw = new RandomAccessFile(ft,"rw");

   int i=0,bc=0,k,uc=0,d=3;
   long Nc=0;
   byte ch,code;
   byte triplet[]=new byte[d];
   char s[]=new char[d];

   ch=(byte)fr.read();

        while(ch!=-1)
         {
         if(ch=='A' || ch=='C' || ch=='T' ||ch=='G' || ch=='a' || ch=='c' ||
ch=='t' || ch=='g')
                {
                triplet[i]=ch;
                s[i]=(char)ch;
                i++;bc++;

                   else if(ch=='N')
                   {
                   for(k=0;k<bc;k++)
                   fw.write(triplet[k]);
                    Nc=1;
                   while((byte)fr.read()==(byte)'N')
                   {Nc++;}
                   fw.write((byte)'N');
                   fw.writeBytes(String.valueOf(Nc));
                   Nc=0;
                   fr.seek(fr.getFilePointer()-1);
                   triplet=new byte[3];
                   bc=0;i=0;
                   }

                  if(bc==d)
                  {
                   textField13.setText(String.valueOf(s));
                   VAR_LUT(triplet,d);
                   textField14.setText(String.valueOf(w));
                   if(found==0)
                   {
                    fw.writeByte(dw);
                    textField15.setText(String.valueOf((char)dw));
                   }

                   triplet=new byte[d];
                   s=new char[d];
                   bc=0;i=0;
                   }

              }

          ch=(byte)fr.read();
          g=(g+f);
          if(g>100)
             g=100;

      textField17.setText(String.valueOf((int)g));

      }

       if(bc!=0 || found==0)
       {
                   code=search_data_code(w);
                   fw.writeByte(code);
                   textField15.setText(String.valueOf((char)code));

          for(k=0;k<bc;k++)
          {
          fw.write(triplet[k]);
          textField15.setText(String.valueOf((char)triplet[k]));
          }
       }

      fr.close();
      fw.close();
   }

 catch(Exception e)
{
   System.out.println("compress Button "+e);
}

 textField7.setText(String.valueOf(ft.length())+" Bytes");
 Long v=100-(ft.length()*100/fs.length());
 textField16.setText(String.valueOf(v)+"%");
}

public void VAR_LUT(byte triplet[],int d)
{
    found=0;
    byte b[];
    String s1,t;
    char c[]=new char[d];
    int p;

    for(p=0;p<=(d-1);p++)
    c[p]=(char)triplet[p];

    t=String.copyValueOf(c);
    wt=w+t;

     try
     {

     String str="SELECT * FROM
2D_VARIABLE_LENGTH_LUT";
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     Connection
con=DriverManager.getConnection("jdbc:odbc:dna_ds_dd");
     Statement stmt =
       con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
       ResultSet.CONCUR_UPDATABLE);
      ResultSet rs = stmt.executeQuery(str);

          rs.first();
          while(rs.isAfterLast()==false)
          {
           s1=rs.getString("data_word");
             if(s1.equals(wt))
            {
               w=wt;
               found=1;
               break;
             }

          rs.next();
          }

    if(found==0)
    {
         byte rb2;
         rs.first();
         rb2=(byte)rs.getInt("code");
         rb2=(byte)(rb2-1);

      if(rb2>-128)
      {
      rs.moveToInsertRow();
      rs.updateString("data_word",wt);
      rs.updateInt("code",(int)rb2);
      rs.insertRow();
      }

      dw=search_data_code(w);
      w=t;
   }

      rs.close();
      con.close();
 }

     catch(Exception e)
     {
        System.out.println("VAR_LUT "+e);
     }
}

public byte search_data_code(String w)
{
   byte code=-1;
   byte b[];
     try
     {
     String str="SELECT * FROM
2D_VARIABLE_LENGTH_LUT";
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     Connection
con=DriverManager.getConnection("jdbc:odbc:dna_ds_dd");
     Statement stmt =
       con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
       ResultSet.CONCUR_UPDATABLE);
      ResultSet rs = stmt.executeQuery(str);

          rs.first();
          while(rs.isAfterLast()==false)
          {
           String s1;
           s1=rs.getString("data_word");

               if(s1.equals(w))
               {
                 code=(byte)rs.getInt("code");
                 break;
                }

               rs.next();
          }

      rs.close();
      con.close();
     }

     catch(Exception e)
     {
        System.out.println("search data code "+e);
     }
  return(code);
}
```

De-Compress File Button Code for 2D Coding With Variable Length LUT

```
private void
jButton8ActionPerformed(java.awt.event.ActionEvent evt)
 {
   String tf;
   String rsym=null;
   long size;
   float f,g=1;

   tf=textField2.getText()+"/"+textField3.getText();
   File fs=new File(textField1.getText());
   File ft=new File(tf);
   textField6.setText(String.valueOf(fs.length())+" Bytes");
   size=fs.length();
   f=100.00f/size;
   try
   {
   RandomAccessFile fr = new RandomAccessFile(fs,"r");
   RandomAccessFile fw = new RandomAccessFile(ft,"rw");
   byte ch;
   String word;

   ch=(byte)fr.read();
          while(ch!=-1)
          {
             textField15.setText(String.valueOf((char)ch));
             if(ch>=0)
             {

                fw.write(ch);
                textField13.setText(String.valueOf((char)ch));
             }
              else
              {
              word=get_word(ch);
              textField14.setText(String.valueOf(word));
              fw.writeBytes(word);
              textField13.setText(String.valueOf(word));
              }
              ch=(byte)fr.read();
             g=(g+f);
             if(g>100)
             g=100;
         textField17.setText(String.valueOf((int)g));
          }

         fr.close();
         fw.close();
   }
    catch(Exception e)
   {
      System.out.println("decompress button 2D_VAR_LENGTH
"+e);
   }
    textField7.setText(String.valueOf(ft.length())+" Bytes");
    Long v=(ft.length()*100/fs.length());
    textField16.setText(String.valueOf(v)+"%");
  }

public String get_word(byte ch)
{
   String s=null;
   byte b[];
   try
    {
    String str="SELECT * FROM
2D_VARIABLE_LENGTH_LUT";
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection
con=DriverManager.getConnection("jdbc:odbc:dna_ds_dd");
    Statement stmt =
      con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
     ResultSet rs = stmt.executeQuery(str);
    if(rs.next()==true)
     rs.first();
    while(rs.isAfterLast()==false)
    {
       b=new byte[1];
       b[0]=(byte)rs.getInt("code");
       if(b[0]==ch)
       {
         s=rs.getString("data_word");
         break;
       }
       rs.next();
     }
    con.close();
   }
   catch(Exception e)
   {
      System.out.println("get_triplet "+e);
   }
 return(s);
}
```