

Optimised Cost Considering Huffman Code for Biological Data Compression

Classical Huffman code has widely been used to compress biological datasets. Though a considerable reduction of size of data is obtained by classical Huffman code, yet, more efficient encoding is possible by treating binary bits differently considering requirement of transmission time, energy consumption, and likewise. A number of techniques already modified Huffman code algorithm to obtain optimal prefix-codes for unequal letter costs in order to reduce overall transmission cost (time). In this paper, we propose a new approach to improve compression performance of one such extension, cost considering approach (CCA), by applying genetic algorithm for optimal allocation of the codewords to the symbols. The approach starts its operation by generating a set of codewords for different input symbols based on their frequencies by considering cost (length) of 0 and 1 as α and β , $\alpha < \beta$. This step ensures optimal cost encoding, however, compression performance is not better than classical Huffman code. The idea of the proposed approach is to sacrifice some cost to minimise the total number of bits, hence, the genetic algorithm works by giving penalty on the cost. The performance of the approach is evaluated by applying it to compress some standard biological datasets. The experiments yield that the proposed approach improves the compression performance of the CCA considerably without increasing the cost significantly.

Keywords: Data Compression, Huffman Code, Genetic Algorithm, Cost Considering Approach, Data Communication.

1 Introduction

In the recent years, application of battery-powered portable devices, e.g. laptop computers and mobile phones has increased significantly. Proper representation of digital data and their transmission efficiency has become a primary concern for digital community because it affects the performance, reliability, and the cost of computation in both portable and non-

portable devices. CMOS technologies were developed in order to reduce the power consumption both in data processing and transmission. In order to increase transmission speed and reduce transmission cost, parallel data transmission methods are widely used. However, parallel transmission is limited to short distance communications, e.g. locally connected devices, internal buses. Ruling out the possible availability of parallel transmission links over long distance, we are left with its serial alternative only. If we attempt to transfer big files, e.g. DNA sequences, over a serial transmission link then it would take a significant amount of time. However, we cannot overlook this problem because at present parallel processing is widely used to increase throughput and in parallel processing architecture, processing units are usually distributed in different physical locations and task sharing is a must in such architecture.

Data encoding techniques came into action to improve the data transmission efficiency over serial communication medium by compressing data before transmitting. Efficiency can be measured in terms of incurred cost, required storage space, consumed power, time spent and likewise. Data must be encoded to meet the purposes like: unambiguous retrieval of information, efficient storage, efficient transmission and etc. Let a message consist of sequences of characters taken from an alphabet Σ , where $\delta_1, \delta_2, \delta_3, \dots, \delta_r$ are the elements that represent the characters in the source Σ . The length of δ_i represents its cost or transmission time, i.e., $cost(\delta_i) = length(\delta_i)$. A codeword w_i is a string of characters in Σ , i.e., $w_i \in \Sigma^+$. If a codeword is $w_i = \delta_{i1}, \delta_{i2}, \dots, \delta_{in}$, then the length or cost

of the codeword is the sum of the lengths of its constituent elements:

$$\text{cost}(w_i) = \sum_{j=1}^n \text{cost}(\delta_{ij}) \quad (1)$$

If all the elements of a codeword has unit cost or length then the cost of the codeword is equivalent to the length of the codeword. However, it is not necessary for the elements in the codeword to have equal length or cost. For example, in Morse Code all the ASCII characters are encoded as sequence of dots (·) and dashes (–) where a dash is three times longer than a dot in duration (Redmond, 1964). However, the Morse code scheme suffers from the prefix problem (Grunwald and Vitany, 2003). Ignoring the prefix problem, Morse code results in a tremendous savings of bits over ASCII representation. Using Morse code, we can treat the binary bits differently; 0 as a dot and 1 as a dash. Even if we consider the voltage level to represent the binary digits then they are still different. Table 1 shows the logic level to represent binary digits in CMOS and TTL technologies where V_{DD} is the supply voltage and V_{CC} is 4.75 V to 5.25 V .

Table 1: Example of binary logic level

Technology	0	1
CMOS	0 V to $\frac{V_{DD}}{2}$	$\frac{V_{DD}}{2}$ to V_{DD}
TTL	0 V to 0.8 V	2 V to V_{CC}

As the unequal letter cost problem is not new therefore it has been addressed by different researchers. The more general case where the costs of the letters as well as the probabilities of the words are arbitrarily specified was treated by Karp (1961). A number of other researchers have focused on uniform sources and developed algorithm for the unequal letter costs encoding (Gilbert, 1995; Krause, 1962; Varn, 1971; Altenkamp and Mehlhorn, 1980; Perl et al., 1975). Let p_1, p_2, \dots, p_n be the probabilities with which the source symbols occur in a message and the codewords representing the source

symbols are w_1, w_2, \dots, w_n then the cost of the code W is:

$$C(W) = \sum_{i=1}^n \text{cost}(w_i) \cdot p_i \quad (2)$$

The aim of producing an optimal code with unequal letter cost is to find a codeword W that consists of n prefix code letters each with minimum cost c_i that produces the overall minimum cost $C(W)$, given that costs $0 < c_1 \leq c_2 \leq c_2 \dots \leq c_n$, and probabilities $p_1 \geq p_2 \geq \dots \geq p_n > 0$.

Huffman code (Huffman, 1952) is an efficient data compression scheme that takes into account the probabilities at which different quantisation levels are likely to occur and results in fewer data bits on the average. It is widely used to compress biological data, however, all the techniques use the classical form of the Huffman code where bits are treated equally. Out of many variations of the Huffman code where cost of bits are treated unequally, the most recent approach is described by Kabir et al. (2014). This approach treated binary bit 0 as a dot (·) and 1 as a dash (–) like Morse code and reduces the transmission cost (time) significantly. Like other variations of the cost considering Huffman code, the compression performance (in terms of number of bits require to encode a message) of this approach is not also better than the classical Huffman code. This approach only considers cost reduction but ignores bits reduction therefore number of total bits are considerably high. In this paper, we have proposed a new optimised cost considering Huffman code based on the approach shown in (Kabir et al., 2014). The proposed approach aims at reducing total number of bits by applying a genetic algorithm. The algorithm works by giving penalty on cost to reduce number of bits, but also ensures that the overall cost is lower than classical Huffman code. The efficiency of the method is evaluated by applying it to compress some standard biological datasets. The rest of the paper is organised as follows: Section 2 presents the background study of the issues in biological data processing and the

Huffman code. The proposed approach is described in Section 3. Experimental results and discussion are presented in Section 4. Finally, concluding remarks are presented in Section 5.

2 Background

2.1 Issues in Biological Data Transmission

The size of biological data including DNA sequences increase with an ever expanding rate and will be bigger and bigger in the future. These biological data are stored in biology databases. The exponential growth of these databases become a big problem to all biological data processing methods (Howe et al., 2008). Different operations are applied to these data such as searching (Valentin et al., 2010), e-mail attachment (Scott et al., 2009), alignment (Chenna et al., 2003), and transmission on distributed computing platforms (Tzu-Hao et al., 2014). Interestingly, biological data compression can play a key role in all sort of biological data processing.

A recent deluge of interest in the development of new tools for biological data processing requires efficient methods for data compression. The main objective of data compression methods is minimising the number of bits in the data representation.

Brandon et al. (2009) proposed a new general data structure and data encoding approach for the efficient storage of genomic data. This method encodes only the differences between a genome sequence and a reference sequence. For encoding, the method uses different fixed-length encoding schemes such as Golomb (Golomb, 1996), Elias codes (Elias, 1975) and variable codes such as Huffman codes. There are other methods based on the same idea of encoding only the difference between reference sequence and the target one. One such approach (Scott et al., 2009) uses Huffman code for encoding difference between sequences to sent it as an email attachment. The main limitation of this method is that it must send the

reference sequence for at least once for each species, and usually this sequence is too big to be sent as an email attachment. Wang and Zhang (2011) proposed a new scheme for referential compression of genomes based on the chromosome level. The algorithm searches for longest common subsequence between two sequences and then the differences between them are encoded using Huffman code. All previous studies focus only on the differences and the relation between continuation of the sequence, and use existing encoding approaches to encode biological datasets without considering possible improvement of the encoding schemes.

2.2 Huffman Codes

In computer science and information theory, Huffman code is an entropy encoding algorithm used for lossless data compression. It takes into account the probabilities at which different symbols are likely to occur and results into fewer data bits on the average. For any given set of symbols and associated occurrence probabilities, there is an optimal encoding rule that minimises the number of bits needed to represent the source. Encoding symbols in predefined fixed length code, does not attain an optimum performance, because every character consumes equal number of bits irrespective to their degree of contribution to the whole data. Huffman code tackles this by generating variable length codes, given a probability usage frequency for a set of symbols. It generates prefix-code to facilitate unambiguous retrieval of information. A scheme of prefix code assigns codes to letters in Σ to form codeword w_i such that none of them is a prefix to another. For example, the codes $\{1, 01, 001, 0001\}$ and $\{000, 001, 011, 111\}$ are prefix-free, whereas the code $\{1, 01, 100\}$ is not, because 1 is a prefix in 100.

Applications of Huffman code are pervasive throughout computer science. The algorithm to completely perform Huffman encoding and decoding is explained in (Amsterdam, 1986). It can be used effectively where there is a need for a compact code to represent a long series of

a relatively small number of distinct bytes. For example, Table 2 shows 8 different ASCII characters, their frequencies, ASCII codes and the codewords generated for those symbols using Huffman code. It is seen from the table that the codeword to represent each character is compressed and the most frequent character gets the shortest code. In this example, the compression ratio obtained by Huffman code is 64.16%.

Table 2: Example of application of Huffman Code to compress ASCII characters

Symbols	Frequency	ASCII Code	Huffman Code
A	50	01000001	00
B	35	01000010	101
C	42	01000011	110
D	22	01000100	1001
E	65	01000101	01
F	25	01000110	1111
G	9	01000111	1000
H	23	01001000	1110

There are many other variants of Huffman codes that compress source data to reduce data size and/or transmission cost. For example, Mannan and Kaykobad (2003) introduced block technique in Huffman coding which overcomes the limitation of reading whole message prior to encoding. In the classical Huffman coding scheme, the letter costs are considered as equal. The unequal letter cost versions of Huffman codes scheme are proposed in (Golin and Young, 1996; Golin et al., 2002; Bradford et al., 2002; Golin et al., 2012). In these versions, letters of the alphabet are treated unequally. Recently, in (Kabir et al., 2014) a method is proposed to show the effects of unequal bits cost on classical Huffman code. The idea of this method is to assign the most frequent symbol the minimum cost code and the least frequent symbol the maximum cost code, whereas classical Huffman code assigns most frequent symbol the minimum length code and the least fre-

quent symbol the maximum length code. As a result, the approach produces an optimal prefix-codes for unequal letter cost, thus reduces the overall (transmission) cost of the encoded data.

3 The Proposed Approach

A genome is a stretch of DNA that encodes a polypeptide (protein) which is a set of amino acids bound together in a specific order. Each genomic sequence consists of nucleotides bound together, which are interpreted by the cellular machinery in groups of three, called triplets (Lodish et al., 1999). This is the main reason to divide the whole sequence in a set of triplet and give a code to each triplet. As each DNA sequence contains a combination of four nucleobases—guanine (G), adenine (A), thymine (T), and cytosine (C), therefore it is possible to have $4^3 = 64$ triplets. The first step in the optimised cost considering algorithm is cutting the genome sequence in triplets, then compute the frequency of each triplet in the whole sequence. This table of frequencies are used by the cost considering Huffman code to generate minimal cost code for each triplet (frequency). Finally, these codes with frequencies are used by the optimised cost considering algorithm to generate the optimal allocation with a given penalty on cost. The whole process is shown in Fig.1.

3.1 Cost Considering Approach of Huffman code

The classical Huffman algorithm aims at reducing total number of bits and it constructs a tree in a bottom up fashion. It is shown in (Golin and Rote, 1998) that if the costs of letters are considered unequal then the straightforward bottom up greedy approach does not work. Kabir et al. (2014) uses a top down approach to build a binary tree considering unequal letter cost of bits. They considered cost (length) of 0 and 1 as integer constants α and β , $\alpha < \beta$. Using the analogy of Morse code's ‘.’ and ‘—’,

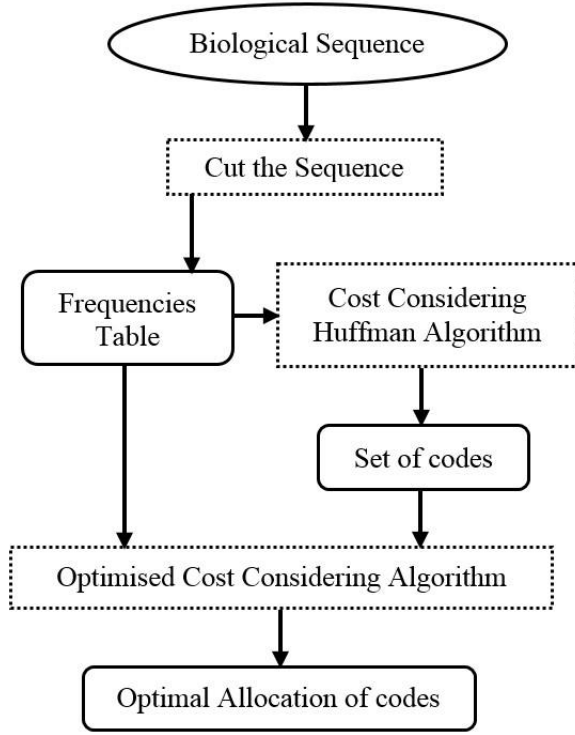


Figure 1: The proposed scheme

the value of α and β is set as 1 and 3 respectively. The complete algorithm to obtain an optimal prefix-free code for unequal letter cost is shown below (algorithm 1). The input to the algorithm are the distinct triplets contained in the genome sequence to be encoded and their frequencies. The process of creating the binary tree starts with a single node (root node) and it is initialised with cost 0. After that, two child (leaf) nodes are created for the root node, i.e., level of the tree is increased by one. Cost of the left child is calculated as the summation of the cost of its parent node and the length of the left arc, and cost of the right child is calculated as the summation of the cost of its parent node and the length of the right arc. Length of left and right arcs are actually the cost (length) of 0 (α) and 1 (β) respectively. The next step is to take a child node with least cost and create two child nodes for it and make it a parent node. In this way, in every iteration the child node with least cost becomes a parent node with two new child nodes. Creation of new child nodes is stopped when total number of child nodes become equal

to the number of distinct triplets needed to be encoded.

Algorithm 1 Cost Considering Algorithm (CCA)

Require: Distinct symbols contained in the message to be encoded and their frequencies

Ensure: Non-uniform / variable letter cost i.e., Cost-considering balanced tree

```

1: for each distinct symbol  $i$  do
2:    $Enqueue(max\_Q, frequency[i])$ 
3: end for
4: create a root node
5:  $cost[root] \leftarrow 0$ 
6:  $Enqueue(min\_Q, cost[root])$ 
7: Define costs of the left and right child of the binary tree
8: repeat
9:    $cost\_of\_parent\_node \leftarrow Dequeue(min\_Q)$ 
10:  create left and right child for this node
11:   $cost[left\_child] \leftarrow cost\_of\_parent\_node + left\_child\_cost$ 
12:   $Enqueue(min\_Q, cost[left\_child])$ 
13:   $cost[right\_child] \leftarrow cost\_of\_parent\_node + right\_child\_cost$ 
14:   $Enqueue(min\_Q, cost[right\_child])$ 
15:  Mark parent node as explored
16: until  $2(n-1)$  nodes are created
17: while  $min\_Q \neq \emptyset$  do
18:    $leaf\_node \leftarrow Dequeue(min\_Q)$ 
19:    $frequency[leaf\_node] \leftarrow Dequeue(max\_Q)$ 
20: end while
21: for each parent node  $j$  do
22:    $frequency[j] \leftarrow frequency[left\_child] + frequency[right\_child]$ 
23: end for
24: repeat
25:   if conflict between nodes then
26:     resolve conflict by swapping conflicted nodes
27:     calculate and reassign cost of all affected nodes
28:     calculate and reassign frequency of all affected nodes
29:   end if
30: until all conflicts are resolved

```

Now the tree T is constructed and the cost of the tree actually depends on how the frequencies are assigned to the leaf nodes. The overall cost will be minimised if the leaves with highest cost always have smaller or equal weight (frequency). To fulfil this condition the leaves of the T are enumerated in non-decreasing order of their cost, i.e., $cost(l_1) \leq cost(l_2) \leq \dots \leq cost(l_n)$, and that $f_1 \geq f_2 \geq \dots \geq f_n$, where l_i and f_i are leaf nodes and frequencies of distinct triplets respectively for $i = 1, 2, \dots, n$. The frequency or weight of parent nodes are calculated as the sum of the frequencies of its child nodes, and it continues upwards until the root node is reached. After that, the algorithm checks for any possible conflicts between all pair of nodes. Two nodes are considered as conflicted if the node with a higher cost has higher frequency violating the above condition, i.e., if $cost(l_i) > cost(l_j)$ and $f(l_i) > f(l_j)$, then there remains a conflict. If there remains a conflict between nodes, then it is resolved by swapping the nodes and recalculating the cost of the tree downward and frequency of the nodes upward. When all the conflicts if existed are resolved then the algorithm generates codes for each of the distinct triplets.

3.2 Optimisation of the Codes

3.2.1 Problem formulation

The problem of finding the best allocation of codes to each triplet can be modelled as an assignment problems, the problem is formulated as follows :

Definition: Given a set of codes $C = \{C_1, C_2, \dots, C_n\}$, and a set of frequencies $F = \{F_1, F_2, \dots, F_n\}$. For each code we have the length of the code $|C_i|$ (number of bits) and the cost of the code S_{C_i} , the objective is to assign to each frequency a code in order to minimise the total number of bits, while respecting the initially assigned total cost S_t with a given penalty $\lambda \rightarrow [0, 1]$. This penalty coefficient represents the allowed amount of cost that can be sacrificed to optimise the total

number of bits.

The Objective Function is to:

$$\text{Minimise } \sum (|C_i| \times F_j) \quad (3)$$

while :

$$\sum (|S_{C_i}| \times F_j) \leq (\lambda + 1) S_t \quad (4)$$

3.2.2 Basic Genetic Algorithm

Genetic Algorithm (GA) is a bio-inspired meta-heuristic algorithm developed by (Mitchell, 1998). GA is a stochastic optimisation algorithm imitate the natural evolution process of genomes. GA started by generating an initial population of random feasible solutions. The optimisation process of GA takes the initial population and generates a new population based on it. The process can be described as follow:

Firstly, select two or more solutions from the current population by using one of the well-known selection techniques (Blickle and Thiele, 1995). These selected solutions will be considered as parents. The second operation of the genetic algorithm is the crossover, which takes these parents as inputs to generate other new solutions considered as children. The third operation of the genetic algorithm is the mutation which ensures a good diversification in the search process. The new solutions can be mutated according to a given mutation probability. The mutation operation changes the value of one or more positions in the solution. The quality of each solution is verified by the fitness function which controls the evolution of the GA population by deletion of the worst solutions and insertion of the good solutions among parents and sons. This processes is repeated until the stopped criteria is reached which can be the number of generation or if the population is stabilised.

3.2.3 Optimised cost considering algorithm

The main objective of the optimised cost considering algorithm (OCCA) is to apply GA to find an optimal allocation of the codewords to

each triplets to reduce the total number of bits. In GA, the encoding of the solution into a chromosome is one of the most important issues in obtaining good optimisation results. The OCCA uses two fixed length arrays of size “64” which is the number of combination for all nucleotides. The first array contains the frequencies of each triplets and the second array contains the cost of the codewords assigned to each triplets. Our genetic algorithm utilises the two arrays and uses the index of each entries in the allocation process.

The genetic algorithm is a stochastic algorithm based on random evolution process. The algorithm 2 describes the outline of the OCCA. Generally the initial population is generated in a random affectation of codes to different triplets. In the OCCA algorithm, the population contain firstly the affectation given by the CCA. The rest of the population are randomly generated, all these generated solutions must satisfy the initial cost constraints, which is the cost (S_i) of the solution produced by CCA and previously added to the population with a give penalty (step 1).

The evolution of the population is the key of the genetic optimisation algorithm. During each generation the process start with the selection of a part (set of solutions) of the population to breed a new generation. In the literature many selection methods have been proposed to guide the population evolution (Blickle and Thiele, 1995). The different methods for selecting a part (set of solutions) of the population vary from a random selection method to heuristic based selection method. We have chosen to select randomly the part of the population to be processed as the heuristic methods are very time-consuming (step 3). After that the operations of genetic algorithm is applied on the initial population to generate a new generation of the population (see Fig.2). Firstly the crossover operation is applied to these two selected solutions (considered as parents) to generate two new solutions (considered as children)(step 4). In the literature many crossover techniques have been used in genetic

algorithm (Osaba et al., 2014), such as one-point crossover which divides the chromosomes into two fragments and recombines the second fragments with the other chromosome’s second fragment. The two-point crossover divides the chromosomes into three fragments and recombines the middle fragment with the middle fragment of the other chromosome. There are many other crossover techniques to allow a good convergence of the algorithm. In our case, we have used the two point crossover with two parameters, the first parameter p_c is a random value in $[0,63]$, which represents the first cut point and the second parameter p_m is also a random value in $[0,63]$, which represents the number of positions to be crossed. These two random parameters are to ensure a good diversification on the whole search space (step 4).

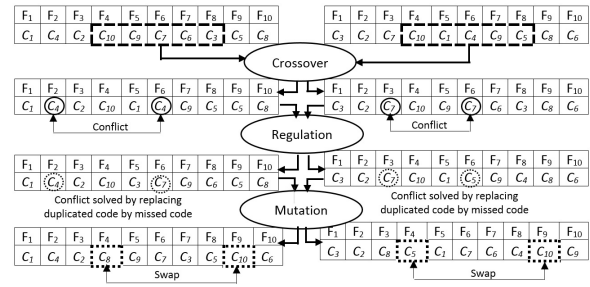


Figure 2: Operations of genetic algorithm

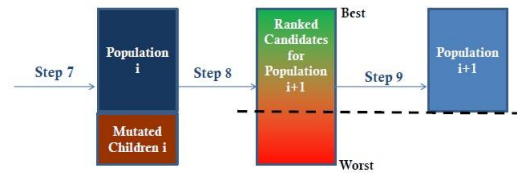


Figure 3: Population update for genetic algorithm

After the crossover operation, the newly generated children may contain conflict, for example, a single code is allocated to two different frequencies in the solution. To overcome such a conflict, a regulation operation is performed to refine the solution to ensure the correctness of the solution (see Fig.2). Secondly these two

Algorithm 2 Optimised Cost Considering Algorithm (OCCA)

- 1: Population initialization (P).
 - 2: **repeat**
 - 3: Select two solutions S_1, S_2 form P.
 - 4: Crossover S_1, S_2 to generate S_{11}, S_{21} (Children).
 - 5: Mutate S_{11}, S_{21} .
 - 6: Validate children with cost constraint (equation 4).
 - 7: Add children to population
 - 8: Rank the population by fitness
 - 9: Remove worst candidates until population limit
 - 10: **until** Max number of generation not achieved
 - 11: Display the best solution from the population P;
-

new solutions are mutated according to a predefined probability γ , the best value of the mutation rate is very problem specific (step 5). In our case, the value of γ is fixed to 0.2 to explorer a few positions in the solution. The mutation operator is used to maintain genetic diversity from one generation of the population of genetic algorithm to the next. In our case, we have used the mutation as a random swap mutation operator (see Fig.3). Each newly generated solution must satisfy the cost constraint. The next step is to add these two new solutions (children) to the current population (step 7) (see Fig. 3). Finally, the new population are ranked by fitness (step 8), and the worst solutions are deleted until the initial size of the population is obtained (step 9). The whole processes are repeated until the maximum number of operation is performed (step 10).

4 Results And Discussion

The effectiveness of the approach has been evaluated with different real genomic biological data, these genomes were downloaded from a recent version of The National Center for Biotechnology Information (NCBI) avail-

able on (<http://www.ncbi.nlm.nih.gov>) (Pruitt et al., 2009). We focused on the sequences alone, ignoring any header and any other exogenous information. In table 3, the different data sets are described with the size of each of them in megabytes (MB) and the references on the biological data bank. Table 4 presents the results obtained by the classical Huffman code, the cost considering algorithm (CCA) and the optimised cost considering algorithm (OCCA) without penalty on cost. The results show that the number of bits required by the classical Huffman algorithm to encode genomic data is the minimum among the other algorithms but the cost is maximum. The cost considering algorithm improves the representation of the generated codes in terms of cost but the number of bits. However, still it compresses the data by 37.54% in the best case, 16.51% in the worst case, and 22.08% on an average. In terms of cost, in the best case the CCA improves cost over classical Huffman code by 12.66%, in the worst case by 11.80%, and on an average by 12.24%. The optimised cost considering algorithm tries to find the best allocation of codes to frequencies by giving penalty on cost. The outcome of this process is a fall in the total number of bits and a rise in the total cost. However, the cost is always lower than the cost incurred by the classical Huffman algorithm. At first, the OCCA optimises number of bits without applying any penalty on the cost (see table 4). Afterwards, it continues giving penalty ranging from 1% to 10% on the cost until a balance is found between total cost and bits.

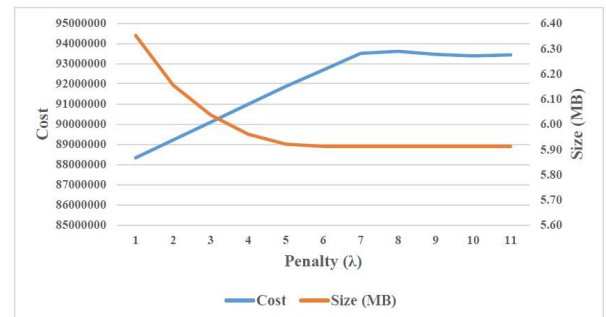


Figure 4: Convergence of OCCA for *Genome 2*

Table 3: Datasets description

Data sets	Name	Size (MB)	Reference
Genome 1	<i>Mycobacterium smegmatis</i>	6.66	CP009496
Genome 2	<i>Amycolatopsis benzoatilytica</i>	8.30	NZ_KB912942
Genome 3	<i>Mycobacterium rhodesiae</i> NBB3	6.11	CP003169
Genome 4	<i>Streptomyces bottropensis</i> ATCC 25435	8.54	NZ_KB911581
Genome 5	<i>Mycobacterium smegmatis</i> str. MC2 155	6.66	CP009494
Genome 6	<i>Mycobacterium smegmatis</i> MKD8	6.76	NZ_KI421511
Genome 7	<i>Bradyrhizobium WSM471</i>	7.42	NZ_CM001442
Genome 8	<i>Amycolatopsis thermoflava</i> N1165	8.27	NZ_CM001442
Genome 9	<i>Bacillus thuringiensis</i> Bt407	5.74	NZ_CM000747
Genome 10	<i>Bacillus thuringiensis</i> serovar thuringiensis	6.03	NZ_CM000748
Genome 11	<i>Pseudomonas aeruginosa</i> 9BR	6.48	NZ_AFXI01000001
Genome 12	<i>Bacillus thuringiensis</i> serovar berliner ATCC	5.97	NZ_CM000753
Genome 13	<i>Bacillus thuringiensis</i> serovar pakistani	5.75	NZ_CM000750
Genome 14	<i>Pseudomonas aeruginosa</i> LES400	6.28	CP006982
Genome 15	<i>Mus musculus</i> chromosome 1	25.58	GL456087
Genome 16	<i>Danio rerio</i> chromosome 1	56.14	CM002885
Genome 17	<i>Homo sapiens</i> chromosome 18	76.64	CM000680
Genome 18	<i>Homo sapiens</i> chromosome 22	99.94	CM000684

Table 4: Comparison of performance among classical Huffman code, CCA, and OCCA without penalty

	Huffman Algorithm		CCA		OCCA ($\lambda=0$)	
Data sets	Cost	Size(MB)	Cost	Size(MB)	Cost	Size(MB)
Genome 1	76787151	4.44	67416213	4.92	67416213	4.82
Genome 2	100425402	5.81	88430665	6.51	88430665	6.35
Genome 3	75940155	4.39	66745619	4.87	66745619	4.79
Genome 4	103552729	5.96	90821835	6.69	90821835	6.47
Genome 5	82234926	4.74	71963876	5.27	71963876	5.17
Genome 6	83454842	4.81	73038795	5.34	73038795	5.24
Genome 7	92539488	5.36	81416359	5.93	81416359	5.85
Genome 8	99613856	5.74	87102639	6.41	87102639	6.30
Genome 9	71876739	4.15	62998800	4.58	62998800	4.49
Genome 10	75324432	4.36	66084958	4.81	66084958	4.73
Genome 11	80766360	4.66	70620666	5.41	70620666	5.04
Genome 12	74560825	4.31	65359604	4.75	65359604	4.67
Genome 13	71562941	4.14	62758225	4.56	62758225	4.50
Genome 14	78261299	4.51	68354090	4.8	68354090	4.89
Genome 15	324439242	18.77	286008420	20.66	286008420	19.94
Genome 16	703734840	40.77	618859291	45.08	618859291	43.18
Genome 17	901032667	52.08	791840455	57.36	791840455	55.12
Genome 18	983434816	56.98	867299889	62.42	867299889	60.70

Table 5: Effects of the penalty on the compression performance of the OCCA

Data sets	Cost	Size(MB)	λ (%)	Compression ratio %
Genome 1	70760174	4.50	4%	32.72%
Genome 2	92010490	5.91	5%	28.79%
Genome 3	69421783	4.47	3%	26.84%
Genome 4	96294638	5.99	5%	29.85%
Genome 5	74855668	4.81	5%	27.77%
Genome 6	76738330	4.86	4%	28.10%
Genome 7	84667949	5.47	3%	26.28%
Genome 8	92416243	5.76	5%	30.35%
Genome 9	66145783	4.21	4%	26.65%
Genome 10	68751359	4.44	3%	26.63%
Genome 11	72737300	4.79	2%	26.08%
Genome 12	67981988	4.38	3%	26.63%
Genome 13	65896779	4.18	4%	27.30%
Genome 14	71762305	4.55	4%	27.54%
Genome 15	297188002	19.31	3%	24.51%
Genome 16	643785655	41.68	3%	25.75%
Genome 17	823506882	53.05	3%	30.78%
Genome 18	901515198	58.99	3%	40.97%

Fig. 4 shows the convergence of the OCCA for minimising number of bits for genome 2 according to the cost constraint. For each genome, a maximum amount of effective penalty is identified, after this maximum value, increasing the penalty is not helping any more to reduce number of bits, i.e., number of bits reach to minimum and cost to maximum. The table 5 presents the best founded number of bits for different datasets with different effective penalty. As seen in the table, the OCCA improves the compression ratio from 37.54% to 40.97% in the best case, from 16.51% to 24.51% in the worst case, and from 22.08% to 28.53% in the average case. This is evident from the table 5 that this improvement is obtained without increasing the cost significantly.

5 Conclusion

In this paper, we have discussed how the cost considering algorithm (CCA) of Huffman code (Kabir et al., 2014) can be used to efficiently compress biological datasets to facilitate efficient data transmission. The advantage of CCA is that it significantly improves (transmission) cost over classical Huffman code given that the cost of bits are different, however, it takes more bits than the classical Huffman code to encode data. In this paper, we identified genetic algorithm as a potential method to optimise the total number of bits in the CCA. The proposed optimised cost considering approach (OCCA) starts its operation by generating a set of optimal prefix-free codewords for a set of distinct input symbols by applying CCA which treats binary bits unequally.

After that, the approach utilises genetic algorithm to find an optimal allocation of generated codewords to different symbols to reduce the total number of bits require to encode the dataset. In this allocation process the OCCA gives penalty on the cost to improve the compression performance.

The proposed approach is applied to biological genomic datasets and a performance comparison is made with the standard Huffman

code and the cost considering algorithm of the Huffman code. The results show that the proposed approach improves the compression performance of the CCA significantly without affecting the cost notably. In future, we hope to extend this work by reducing number of switching activity between the codewords to further reduce the cost of transmission.

References

- D. Altenkamp and K. Mehlhorn. Codes: Unequal probabilities, unequal letter costs. *Journal of the Association for Computing Machinery*, 27(3):412–427, 1980.
- J. Amsterdam. Data compression with huffman coding. *BYTE*, 11(5):98–108, 1986.
- T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms, 1995.
- P. Bradford, M. J. Golin, L. L. Larmore, and W. Rytter. Optimal prefix-free codes for unequal letter costs: Dynamic programming with the Monge property. *Journal of Algorithms*, 42(2):277–303, 2002.
- M. C. Brandon, D. C. Wallace, and P. Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25(14):1731–1738, 2009.
- R. Chenna, H. Sugawara, T. Koike, R. Lopez, T. J. Gibson, D. G. Higgins, and J. D. Thompson. Multiple sequence alignment with the clustal series of programs. *Nucleic acids research*, 31(13):3497–3500, 2003.
- P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975. doi: 10.1109/TIT.1975.1055349.
- E. N. Gilbert. Coding with digits of unequal costs. *IEEE Transactions on Information Theory*, 41(2):596–600, 1995.

- M. J. Golin and G. Rote. A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs. *IEEE Transactions on Information Theory*, 44(5):1770–1781, 1998.
- M. J. Golin and N. Young. Prefix codes: Equiprobable words, unequal letter costs. *SIAM Journal on Computing*, 25(6):1281–1292, 1996.
- M. J. Golin, C. Kenyon, and N. E. Young. Huffman coding with unequal letter costs. In *ACM Symposium on Theory of Computing*, pages 785–791, 2002.
- M. J. Golin, C. Mathieu, and N. E. Young. Huffman Coding with Letter Costs: A Linear-Time Approximation Scheme. *SIAM Journal on Computing*, 41(3):684–713, 2012.
- S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1996.
- P. D. Grunwald and P. M. B. Vitany. Kolmogorov complexity and information theory: With an interpretation in terms of questions and answers. *Journal of Logic, Language and Information*, 12:497–529, 2003.
- D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. S. Pierre, S. Twigger, O. White, and S. Y. Rhee. Big data: The future of biocuration. *Nature*, 455:47–50, 2008.
- D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, Sept 1952. doi: 10.1109/JR-PROC.1952.273898.
- S. Kabir, T. Azad, A. S. M. A. Alam, and M. Kaykobad. Effects of unequal bit costs on classical huffman codes. In *17th International Conference on Computer and Information Technology*, pages 96–101, 2014.
- R. Karp. Minimum-redundancy coding for the discrete noiseless channel. *IRE Transactions on Information Theory*, 7(1):27–38, 1961.
- R. M. Krause. Channels which transmit letters of unequal duration. *Information Control*, 5(1):13–24, 1962.
- H. Lodish, A. Berk, S. L. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular Cell Biology*. W.H. Freeman and Co Ltd, 4th edition, 1999.
- M. A. Mannan and M. Kaykobad. Block huffman coding. *Computers and Mathematics with Applications*, 46(10-11):1581–1587, 2003.
- M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.
- E. Osaba, R. Carballado, F. Diaz, E. Onieva, I. de la Iglesia, and A. Perallos. Crossover versus mutation: A comparative analysis of the evolutionary strategy of genetic algorithms applied to combinatorial optimization problems. *The Scientific World Journal*, 2014, 2014.
- Y. Perl, M. R. Garey, and S. Even. Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters. *Journal of the ACM (JACM)*, 22(2):202–214, 1975.
- K. D. Pruitt, T. Tatusova, W. Klimke, and D. R. Maglott. NCBI reference sequences: current status, policy and new initiatives. *Nucleic acids research*, 37(suppl 1):D32–D36, 2009.
- W. A. Redmond. International morse code. *Microsoft Encarta 2009 [DVD]*, pages 275–278, 1964.
- C. Scott, L. Yiming, L. Chen, and X. Xiaohui. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, 2009. doi: 10.1093/bioinformatics/btn582.
- C. Tzu-Hao, W. Shih-Lin, W. Wei-Jen, H. Jorng-Tzong, and C. Cheng-Wei. A novel approach for discovering condition-specific

correlations of gene expressions within biological pathways by using cloud computing technology. *BioMed research international*, 2014, 2014.

- F. Valentin, S. Squizzato, M. Goujon, H. McWilliam, J. Paern, and R. Lopez. Fast and efficient searching of biological data resources using eb-eye. *Briefings in bioinformatics*, 11(4):375–384, 2010.
- B. Varn. Optimal variable length codes - Arbitrary symbol cost and equal code word probability. *Information and Control*, 19(4): 289–301, 1971.
- C. Wang and D. Zhang. A novel compression tool for efficient storage of genome re-sequencing data. *Nucleic acids research*, 39(7):e45–e45, 2011.