Compression of DNA Sequences using Context Tree Weighting

Luke Hein Martin - 0926966
Eindhoven University of Technology
l.h.martin@student.tue.nl

***Abstract*— Context tree weighting is a statistical model that assigns a probability estimate for a symbol to occur by weighting estimated probability of all branches in a context tree into a single weighted probability at the root of the tree. This paper aims to access CTW's performance as a DNA compressor. The algorithm is also compared to another compression technique called competitive finite context trees. It is shown that they perform similarly however competitive finite context trees are able to achieve better compression due to the speed in which they can respond to changes in the DNA sequence. Having multiple CTW models of varying depth compete proved to at best provide minimal benefit, and at worst perform worse than a standard CTW model. By also processing the inverted complement of every symbol and context, showed improvements in compression however it came at the cost of increased memory requirements during processing. Lastly a modified version of the CTW method that attempts to exploit properties of codons for better compression was developed and tested. The method performs better for coding DNA, and worse on non-coding DNA than standard CTW. However when attempted to be used as codon classifier the results were inconclusive. However the method still shows potential**

I. Introduction

The modeling of DNA is an important tool in understanding the biology of organism, as it stores its genetic instructions. DNA is composed of four nucleic acids, Cytosine(C), Guanine(G), Adenine(A) and Thymine(T). DNA has a double helix form, where complementary strands of DNA are connected by hydrogen bonds. These bonds occur according to a base pair rules, A only pairs with T and C only with G. In specific regions of DNA, nucleotide triplets called codons. These codons determine the order of amino acids during protein synthesis
Compression of DNA is not only useful for the data management of genomic information, but it is also a useful tool for modeling and learning about DNA. Already compression of DNA has been used to distinguish between coding and non-coding regions of DNA [11], and evaluate the 'distance' between two organism in the evolutionary tree [5]. Given that DNA is known as the code of life, it would be fair to assume that its occurrence is not random and contains some regularities. These regularities may then be exploited for compression. However most general purpose text compression algorithms, and standard tools such as gzip and bzip, are unable to compress DNA below 2 bits per symbol[13][3]. Compression is reliant on the its ability to model the information. By having an accurate model, greater compression is then achievable. So by pursuing greater compression, better models of DNA are also being developed.

II. Background - Extension

In general there are two approaches when it comes to compression, replacement and statistical compression. Replacement compression replaces a repeated subsequence with a pointer to a previous instance or to an entry in a dictionary. DNA can be highly repetitive, so it is natural to use such a method. One common known example is Lempel-Ziv compression [19], the basis for .zip file. Statistical compression however tries to predict what symbol comes next by building a statistical model of the sequence based on the symbols that have been seen before. By having a high probability of the next symbol occurring, high compression is achieved. The first compression technique specifically developed for compressing DNA was BioCompress developed by Drumbach and Tahi[7]. BioCompress took advantage of the repetitive nature of DNA, by Fibonaci coding exact and complementary repeats, using the length and position of the earliest occurrence. Any non-repeated sequence was encoded as a two bit symbol. Biocompress-2[8] is an extension of Biocompress which uses a 2-order arithmetic encoder in the case where no repetition is found. Biocompress however has very large memory requirements, and therefore is unable to compress large sequences. The Cfact[15] algorithm functions in a similar way to BioCompress, however it is a two phase algorithm. First it constructs a suffix tree which finds the longest repeats present in the sequence. Afterwards it performs the encoding. Repeated sequences are encoded using pointers to previous occurrences, and non-repeated are encoded using two bits per base. Approximate repeats was first exploited by the GenCompress[5] algorithm, where a non-complete repetition are encoded using their position and length. The DNACompress[6] algorithm makes use of imperfect repeats, but does so in a two step process. First finding the approximate repeats, and then encoding them based on distances measure. DNAPAck [2] also exploits approximate repeats, but does so using a dynamic program. Some DNA compression algorithms combine the statistical and replacement techniques. Algorithms such as NML Comp[16] and the improved version GeNML[9] does exactly this. GeNML first breaks the sequence into

fixed sized blocks. The algorithm encodes a block by first finding an inexact repeat, and creates a pointer to the previous occurrence, and uses a bit mask to encode the differences between the block and the closest match. This bit mask is encoded using a normalized maximum likelihood probability distribution. GeNML however does suffer from a very poor runtime. There are a number of purely statistical methods for compressing DNA. CDNA by Loewenstern and Yianilos [12], ARM by Allison et al. [1], and the XM algorithm by Cao et al. [4]. In CDNA, a probability estimate of symbols is calculated from approximate matches. Each probability estimate is combined with weights that are learned adaptively. ARM uses the summation of probability of all explanations on how the source was generated[4]. The XM algorithm builds a probability distribution by combining a number of experts, weightings the input of these experts. The weights of the experts are adapted as function of the quality of their recent predictions.

III. The Context Tree Weighting Method

To fully understand the functions of the Context Tree Weighting algorithm, it is important to first understand a number of core concepts. Which are described briefly in the following sections.

A. Sources and Codes

A sequence of length T can be described as follows, $x_1^T = x_1 x_2 ... x_T$ where x_1 is the symbol at position 1 and x_T is the symbol at position T. Given that we are dealing with DNA, x_n can take the value of the characters A, C, G, or T where each character represents their corresponding base. We assume that $P_r(X = A) = \Theta_A$, $P_r(X = C) = \Theta_C$,$P_r(X = G) = \Theta_G$, and $P_r(X = T) = \Theta_T$, where $\theta_{A,C,G,T}$ is the probability of an A, C, G, orT occurring. Therefore this sequence is generated with actual probability $P_a(x_1^T) = (\Theta_A)^{n_a}(\Theta_C)^{n_c}(\Theta_G)^{n_g}(\Theta_T)^{n_t}$ where $n_{a,c,g,t}$ is the number of A's, C's, G's, or T's present in the sequence. [17]

B. One Unknown Parameter

It is not always the case that the corresponding θ parameter is known, therefore it is important to have tools do handle this case. To deal with this unknown parameter, its is possible to use an estimated distribution. One such estimation formulated by Krichevsky and Trofimov[10] is defined as follows:

$$P_e(X_t = s|c) = \frac{n_s^c + \alpha}{n^c + \alpha|\mathcal{A}|} \tag{1}$$

where n_s^c is the number of times before, the symbol s has been seen given with context c, α is scaling parameter, $\mathcal{A}$ is the size of the alphabet from which s can exist. So in this case $\mathcal{A}$ is equal to 4, as used in [17] α is chosen to be 0.5 and where

$$n^c = \sum_{a \in \mathcal{A}} n_a^c \tag{2}$$

Context is the finite set symbols that occurred prior the the symbol s. So for example, given a sequence $GCTCGTA$, the context of order 3 for the last symbol, A, would be TGC, organized by recency when reading from left to right, so the most recently seen symbol is first in the context sequence.

C. Unknown Model - Context Tree

In the case where a model is not known, it is possible to use a *context tree* to compute an appropriate coding distribution. A context tree consists of nodes that correspond to context c up to a certain depth [17]. Each node of the context tree corresponds to the symbol that occurred after the given context up until that point. For example in Fig.1, you can see an example symbol and context pair. The branches of the tree that correspond to the shown context is shown in bold.

D. Finite-context models

A Finite context model assigns a probability to all symbols in an information sources alphabet, using a context tree and probability estimator. So to properly use the KT-estimator as shown in (1), each node of the context tree also need to store counters for each time every specific symbol has been seen given that context. To get an estimation of symbol s with a specific context c, the context c is first used to traverse the context tree. Once at a terminal node, the stored counts are used as inputs for (1) which will return an estimation of s occurring. Once the estimation of the symbol has been calculated the model is then updated. Updating the models means that for every node that was traversed, the counters for the symbol in that node are then incremented accordingly

E. Weighting Alternatives

Given two sources, source 1 and 2, and coding distributions $P_c^1(x_1^T)$ and $P_c^2(x_1^T)$ are good coding distributions for each sources respectively, then the weighted distribution

$$P_c^w(x_1^T) = \frac{P_c^1(x_1^T) + P_c^2(x_1^T)}{2} \tag{3}$$

is a good coding distribution for both sources 1 and source 2 [17].

F. The Context Tree Weighting Method

With all those aspects known, it is now possible to properly formulate the context-tree weighting method. Using a context-tree with depth D to create the coding distribution, also storing the number of times each symbol has been seen per context. Furthermore at each node a weighted probability of node n is stored. Knowing that that the KT-Distribution estimator described in (1) is a good estimation for probability of a symbol occurring. The weighted probability is then calculated as such if the depth(n) $<$ D,

$$P_w^n = \frac{P_e^n + P_w^{C_a} \cdot P_w^{C_c} \cdot P_w^{C_t} \cdot P_w^{C_g}}{2} \tag{4}$$

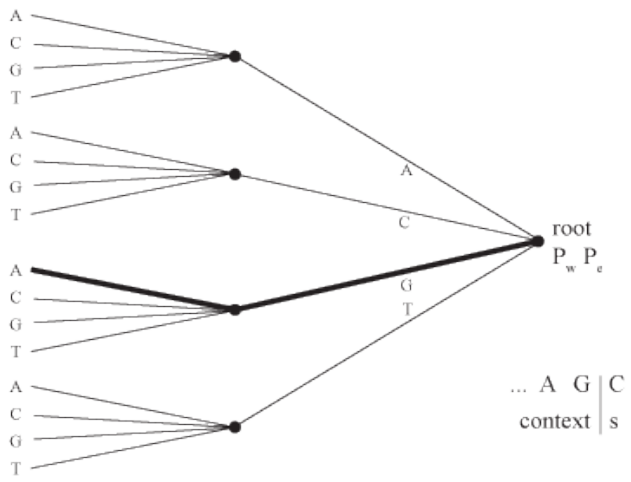

Fig. 1. A context tree with an example symbol and context. The context for the shown symbol is highlighted.

where $P_w^{C_s}$ represents the weighted probability the s child. and

$$P_w^s = P_e^n \tag{5}$$

if depth(n) = D. From what was described earlier about weighted probabilities, it can be assumed that this combination of probabilities is a good approximation. In the case where the node is at max depth, the are no children so the probability at that node is solely determined by the estimated probability P_e

When reading information using the context tree weighting method, it possible to imagine it as *sliding* across the series of symbols. Where the previous D symbols before the symbol currently being considered make up the context. From there the context-tree is traversed according to the context. Where for each node a P_e is calculated using the counts stored in the node. These P_e values propagate up to the root node where its weighted probability (P_w) is updated, and can be used as coding probabilities for arithmetic encoding and decoding. It should be noted that the P_w and P_e is persistent, meaning that time a new symbol is added the existing P_w and P_e are multiplied with the updated version.

$$P_{w_{new}} = P_{w_{old}} \cdot \frac{P_e^n + P_w^{C_a} \cdot P_w^{C_c} \cdot P_w^{C_t} \cdot P_w^{C_g}}{2} \tag{6}$$

Once the estimations of the symbol probability has been calculated, the counts of each node traversed are updated accordingly. [17]

G. Competitive Models

DNA is non-stationary, it changes of the length of the sequence. At times it may be more repetitive, and other times less. That is why many algorithms switch between multiple compression techniques to try account for the changing nature of DNA. This changing aspect of DNA can also be accounted for using CTW and Finite Context Model, however this is done by changing the order/depth of the model. Specifically multiple finite context models and weighted context tree models of varying depths, compete. This approach is based off of the work in [14]

This is done by first breaking the sequence into non overlapping blocks of a fixed size. After each model has been updated using all the symbols within the block. The best performing model is then selected to encode the information. If an encoder/decoder was also included in the implementation, the choice of depth would also have to be communicated. Although only one model is chosen, all models are updated according to the symbols seen in the block. No information is reset between blocks including the context. So the first symbol of the block has its context defined by the symbols in the previous block.

IV. Inverted Complements

The complement to a base is its bonding pair. So T and A are complementary to each other and so is C and G, because it is the base that is present on the other side of the DNA strand. DNA repeats, however sometimes these repetitions occur in the complementary strand. So it is useful to have a statistical model that accounts for this. That is why, when desired, not only are the counts incremented for the symbol that has been encoded, but also for its reverse complement. That reverse complement is constructed as follows. If we were to consider a the symbol A with context $CGAGAT$, where the symbol at the at the beginning of the string is the most recently seen symbol. The two are first combined, resulting in $ACGAGAT$. Now complementing this string (A to T and C to G) results in $TGCTCTA$ now reversing this string results in $ATCTCGT$. From here the final symbol, A,is processed with context $TCTCGT$.

V. Implementation/Methods

A large portion of this Bachelor End Project was developing an implementation of the CTW method in Python. The implementation had to handle trees of depths of up to sixteen, store the information of up to 400,000 symbols and be able to output the changes of probability changes as symbols were being added. There were three types of compression methods that needed to be implemented. The first being a standard CTW model, the second a codon-capable version which manages three CTW, and third a competitive finite context model.

A. Classes & Data-Structure

The implementation primarily makes use of two classes, **node** and **tree**.To create the context-tree information structure that both the finite context and CTW models require, a tree data structure is constructed using a number of **nodes** which all point to one anther in a parent-child manner. The tree structure starts at the root node, which has pointers that are stored in a

python dictionary to each of its children, each node also stores a pointer to its parent node, however the root would not contain one since it has no parent. These pointers allow for the easy traversal of the tree. Each node also stores a set of counters, stored in a dictionary, of the amount of times each symbol has been. Since these counters are stored at the node level, they represent the amount of times a symbol has been seen with the context necessary to reach that node. Each node also stores an estimated probability P_e and a weighted probability P_w. To avoid any issues with underflow, these values are stored as $\log_2$ values. This also means that multiplication of probabilities can be done with simply adding, which is a much less computationally intensive task.

The **tree** class is what manages this tree structure. It updates and maintains the tree's information.

B. Traversing & Updating the Context Tree

Both the finite context and CTW models function on being able to traverse the context tree and then update it once a symbol has been processed. Given the tree structure explained previously, traversing the digital context tree is as simple as looping through, in order, the context of the symbol that needs to be processed, and passing the symbols, in order starting from the root, into the dictionary stored at each node to reach the correct leaf node. If the function finds that it a node that it is trying to access is not available it will create one. Once the function that traverses the tree has reached a leaf node, its returns a list of all the nodes it accessed, ordered with the node deepest in the tree first. This is important for when calculating the P_w values, because it relies on the P_e values of the node's children. This means that they have to updated before their parent node. However the actual calculation of the P_w presents an specific challenge, it requires the addition of two probability values, which when working with log_2 values is not as simple as it may seem. We would not want inverse log and then add as that will defeat the purpose storing the values with log_2. So instead, using

$$\log(p' + p'') = \log p' + \log(1 + 2^{(\log p'' - \log p')}) \quad (7)$$

it is possible to add to values using their log values[18]. However this equation relies on the fact that $p' > p''$. Trough a trail-and-error it became clear that, if p' was the multiplication of the weighted probability of the children nodes, and p'' was the P_e of the node, no errors would occur.

With all this in place, it is now possible to obtain a probability estimate for a symbol with a specific context. First the tree is traversed following the context of the symbol. This will reach a leaf node, where the P_e is updated using the counters stored at that node. Since at this node there are no children, it is also the the P_w. Once this calculation has been done, the counters are incremented. Now following that list of accessed nodes upwards, the same process occurs. First the P_e is updated using the stored counts and then P_w. This continues until reaching the root node, where the final calculation and update is made. Outputting the data for the graphs was done by monitoring the P_w of the root. By finding the difference between the value before, and then after processing either a symbol, or block of symbols, contains the probability the model assigned to predicting that symbol/sequence.

Finite context model function the same as the CTW method, however it does not weight all estimated probabilities in the context tree. This means that only the nodes at the maximal depth need to be considered when finding the predicted probability of a symbol or sequence. So once traversed to a leaf node, the change of P_e in that node is what can be used to encode that symbol. However when accessing the change in probability for a sequence, the total change in all the nodes at maximal depth is what determines the probability the model assigns to that sequence occurring. Calculating this would be costly endeavor since every node needs to be check both before and after the sequence has been processed. This means that for a tree of depth 12 there could potentially be roughly $8 * 10^{12}$ nodes to check, of course that would never happen, but it does show immense amount of nodes that would need to be checked. So instead of doing it that way, a single value was stored that contained the multiplication of the P_e values for the terminal depth. Since all that needs to be known is multiplication of all the probabilities at at a specific depth, this can be stored as a property of the tree.Since the P_e value is stored as $\log_2$ value, this means that it is really a summation of all the values:

$$P_{e_{total}} = \sum_{n=0}^{N} P_e^n \quad (8)$$

where N represents the total number of nodes in the tree at maximal depth. So each time a a node is updated this value can be updated, and is done so in the following way. To ensure that the same node is not added twice, each time a node is updated its P_e value is first subtracted from the $P_{e_{total}}$, and then once it has been updated, added back in again. By doing this the total number of calculations required is only equal to the amount of symbols in the sequence. Now, to measure the probability of a sequence you only have to calculate the difference of the summed value before and after the sequence has been processed.

In [14] it is proposed that for nodes at depth greater than 11, the α in (1) be set to 0.05 instead of 0.5. Since the finite context models are based off of the approach explained in that paper, α was set to 0.05 for finite context models of depth greater than 11.

C. Competitive Models

An idea was introduced earlier about have multiple finite context models and CTW models compete. The key thing to note here is that all the necessary information for a finite context model or CTW model depth d is contained within the a CTW or finite context model of depth $d + 1$ Therefore a CTW model of depth 16 already contains the necessary for models of depth 1 to 15. For the finite context model this is a simple as not only maintaining the summed value of all P_e values at the maximal, but for all depths. For CTW this is a bit tricker, since the P_w are connected, so the P_w for a model of once depth cannot be used in a model of a different depth. However this can be solved by storing all possible P_w per node. For example a node at depth 3 in a tree of total depth 16 has a P_w for depths 3 to 16 and a node at depth 16 only has a P_w for depth 16. Calculating these P_w is done by running a loop over all the stored P_w and updating them using the P_w of the same depth of its children nodes. The competition portion is simply seeing which model performed best, so which ever had the highest probability estimate for the sequence or symbol occurring.

D. Codon CTW

As briefly introduced before hand there is a specific region of DNA known as coding DNA, where triplets of nucleotides called codons are present. It is an important task in biology to know where these coding regions begin and end. Special methods using compression have been developed to find these regions [11]. Given that codons come in triplets it could be fair to assume the bases at the same relative positions are related in some manner, this relation therefore could be used to compress them. The idea was then to encode the symbols of a codon into separate CTW trees.

Three trees would be used, one for each symbol in the triplet. A separate class in the python implementation was added that would manage three separate trees, it maintained the same functionality as the **Tree** class, but rotated through the trees as symbols were added. So if the the first symbol was processed by Tree 1, the next would be processed by Tree 2, the next by Tree 3 and then the next one would again be processed by Tree 1. The context of each symbol was maintained, so if a A was processed with context $CGAT$ the following symbol would have context $ACGA$. Since the probability estimates for a sequence is distributed over three trees, to calculate the P_w of a whole sequence the average difference across all three trees is used.

VI. Results & Discussion

Each DNA sequence that was processed using only the CTW method and Finite Context method, was done using context trees with depths of 1 through 16 for the finite context method, and with a weighted context tree of depth 16.

The first 400,000 well defined symbols of each sequence were processed in blocks of 200 symbols. The block-size and amount of symbols processed was chosen as it was the same size used in [14] which is the basis for the competing finite context models. Furthermore, unless explicated stated, the inverse complement was not added.

Each model performances was judge by measuring the average code word length (CWL) per symbol, which is a measurement of how many bits would be needed to encode that symbol. By plotting the average CWL per symbol, against the symbol number, it is possible to view how the model performed over the length of the sequence.

A. CTW vs Competing Finite Context

To compare the performance of the competing finite context model and the CTW method, both methods were used to process the NW_004929428.1 sequence. Fig.2 shows how the two models performed as more blocks were added were processed. There are a number of interesting features of this graph. First is the rather sporadic codeword lengths. There are some peaks were the model performs poorly, particularly for the CTW method, and there some deep troughs where the models perform very well. The finite context model has a general 'spikiness' to it where throughout the sequence the models performs significantly better than average for a single or very few blocks. This could possibly be explained by local repetitions. Short areas of the DNA where with short repetitive subsequences. It is worth noting the CTW method, although not clear from this Fig.2, also has these same moments of increased performance, this is more clear in Fig.3. However the moments are not as effective as the finite context models. This could be due to the different α used in the finite context tree for depth greater than 11. However after running a test using a CTW model where α was set in the same manner as the the finite context models, although showing some improvement, the finitie context model still outperformed the CTW model.

Given that both models are using the same probability estimator, this should then be related to to the weighted aspect of the CTW method. Perhaps when the model has some kind of statistical inertia, where it is slower to react to the changing aspects of the DNA than the finite context model. This could be due to symbols that have relatively unique contexts for that subsection, propagate the relatively low probability value up the tree.

For both models there is a particular section of the sequence around the 900th block or 180,000 symbol, where both models are able to achieve significantly shorter codeword lengths of down to 0.75 bits in the case of the finite depth models. By looking directly at the sequence, it is abundantly clear that in this area, there is a large number of repetitive sequences.

It is also possible to get an overall sense of how the models performed by taking the average codeword length for the entire processed sequence. For the CTW method,

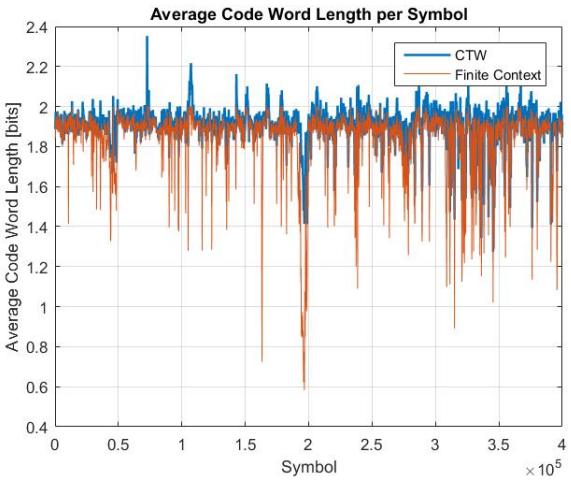

Fig. 2. Graph showing the average code world length of both the CTW method and Competing Finite Context.

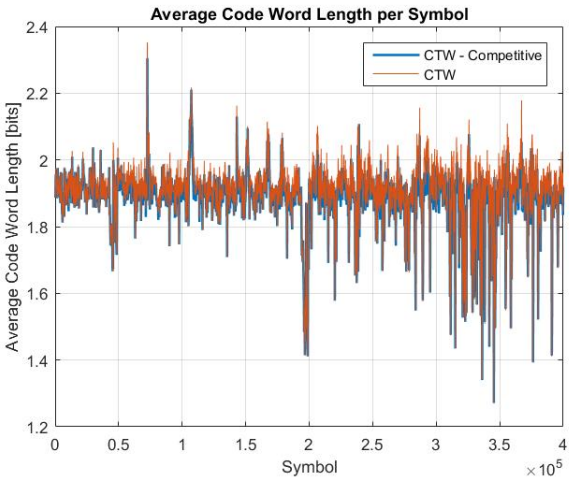

Fig. 3. Graph showing the average code world length of the CTW method against a competitive CTW model.

the overall average code word length was 1.9046 bits and 1.8426 bits for the finite context. However it is also necessary to add 0.02 bits, because it is also necessary to add in the depth choice for the block. 0.02 bits need to be added because 4 bits are needed to represent the depth choice and distributed over all symbols in the block ($\frac{4}{200}$) comes out to 0.02 bits per symbol. When using a CTW model that has α set like the finite context models, the average code world length was 1.8866 bits, a 0.018 bit improvement over normal CTW, but is still out performed by 0.0240 bits by the finite context model.

B. CTW vs Competitive CTW

The method of using competing models of varying depth could also be applied to the CTW method. So taking the weighted probability of trees of vary depths, and selecting the best code word length. Fig.3 shows the results of processing the same set of symbols, again the NW_004929428.1 sequence, through a weighted context tree of depth 16 and a set of competing weighted context models of depths 1 to 16. In theory, since the competing model contains the a depth 16 weighted context tree, it can only perform as good, or better. This is actually what is seen in Fig.3. The competitive model is able to achieve a more consistent code word length, with less peaks and greater performance in areas of local repetition. When considering the previous proposition that there is a certain amount of "inertia" that has to be overcome before the model begins to adapt, it possible then that at lower depths the model is more ready to adapt.

Overall, the competitive model resulted in a total average code word length of 1.8867 bits, and the CTW model resulted in a total average of 1.9046 bits. Again it is necessary to add 0.02 bits to the competitive model, results n 1.9067, which means that it would actually perform worse during encoding. So the benefit provided by having the models compete is undercut by the cost of having to include the depth information. However this is not the case for all sequences that these two models were tested on., the competitive model provided a benefit of 0.0151 bits, for sequence NW_004929429.1, 0.0203 bits for sequence NW_004929430.1 and 0.0233 bits for sequence NW_004929430.1. Overall though the benefit is very minimal at best, and at worst can decrease the performance.

C. Inverted Complement

As introduced earlier, DNA may have repetitions in its complementary strand. Which means that having a model that takes this into consideration, could lead to a model that could perform better. To test this, the NW_004929428.1 DNA sequence was processed by CTW model that did take inverted complements into account, and by one that did not. The results of this processing is visible in Fig.4. It should be noted that only the first 200,000 symbols were taken into account because when processing the inverted complement into account, roughly twice as much information needs to be stored, so adding 400,000 symbols was not possible

It shows what was expected, by taking the inverted complement into account, the model was able to find areas of repetition. For example there are number of points between blocks 400 and 600 where the model takes the inverted complement into account performed significantly better. Overall, when averaging across all blocks, the code world length of the normal CTW model is 1.9104 bits and 1.8995 for the one that takes the inverted complement into account, so 0.0109 bits of better compression. However this does come at the cost of having greater memory requirements, effectively halving the number of symbols that can be processed, as twice as much information needs to be stored for every symbol processed. However the simplicity of the method makes it very easy to implement.

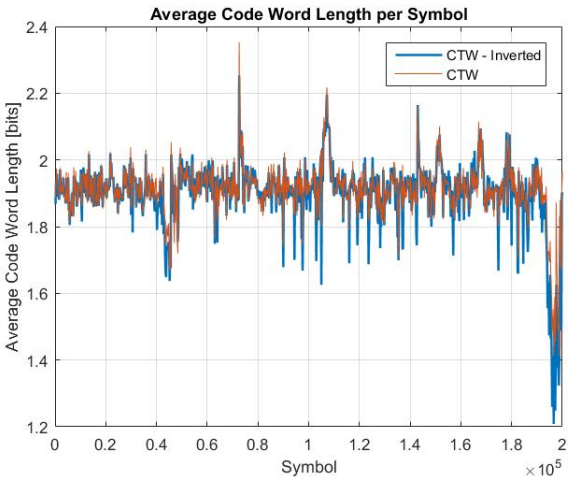

Fig. 4. Graph showing the average code world length of the CTW method against a CTW model that also includes the inverted complements.

D. Codon Regions - Extension

As explained previously, a specific version of the CTW algorithm was prepared attempt to exploit the structure of codon DNA. So to test the performance of this type of model, it was compared against a standard weighted context tree model. For these experiment as many symbols as could fit within a multiple of the chosen block size of 300, was processed. The block size was chosen to be 300 so that all three trees of the codon CTW model would process an equal number of symbols.

To first test whether the underlying assumption was true, the codon adjusted CTW model and a normal CTW model processed a DNA sequence of purely coding DNA (ENST00000589042.5). The results of which can be seen in Fig.5. The graph shows a clearly that the codon version of the CTW method consistently outperforms the standard CTW model. Another experiment was run to see how the codon CTW model performs on non-coding DNA. The results showed that the codon CTW model would consistently perform worse than the normal CTW model.

So it appears then that there is some connection between the position of the symbols in the codon, and that this can be exploited for better compression. However there is a short section at the end of the sequence where the codon model begins to perform significantly worse, it is not clear exactly why that is. Overall the average code world length for the entire sequence for the standard CTW model was 1.9039 and 1.8320 for the codon CTW model. So a 0.0719 bit decrease in code word length.

This is a potentially very interesting result, because this means that this codon CTW model could be used to detect coding regions in DNA. Given that it would perform better on coding regions, and worse on non-coding regions. So if it would be compared to a normal CTW model, in areas where the compression was greater, those could potentially be coding regions, and in areas where the compression was worse, would be regions of non-coding DNA. So to test this, a DNA sequence that had both a mix of coding and non-coding regions was processed. Specifically the NG_027688.1 sequence, the result are shown in Fig.6. Looking closely at the graph it shows that in the beginning of the sequence the two models perform roughly equally, with shorts areas where the normal model performs slightly better, and then in the middle the CTW model begins to out perform the codon model, and vice versa in short regions afterwards. This is perhaps what would have been expected, different regions in which the two models perform better or worse, however this coding regions of this sequence are already known, and when comparing those regions to the graph, the areas do not align. This means that the method, in its current state, was unable to discern between coding and non-coding regions for this specific sequence. Perhaps with some further experiments and change to either model, it could be able to detect coding regions. However that was not in the scope of this bachelor end project.

VII. Conclusions

This project investigated the performance of the context tree weighting algorithm relative to competing finite context models as presented in [14]. It became clear after multiple tests that the finite context model is able to achieve better compression, potentially due able more quickly respond to changes a DNA sequence, both due to its lack of weighting and ability to choose from multiple context depths. Comparing the performance of competing CTW models and a single model of the greatest depth, showed the having competing models provided at times slightly better compression, and at worse a slight decrease in compression due to the necessity to also encode the depth used per block. It was shown that also processing the inverted complement into a CTW model allowed for the finding of palindrome repeats which provided better compression both in subsections of a DNA sequence, and overall. It does come at the cost of having greater memory requirements.

As part of the extension to the bachelor end project a CTW model was made that attempted to exploit some characteristics of the codons, specifically the location of the symbols in the triplet. By having all the symbols in the same position process by the same model, it was thought to that greater compression could be achieved. Although this was true when processing sequences of only coding DNA, the potential for the model of as a coding region detector proved to be inconclusive.

References

[1] Lloyd Allison, Timothy Edgoose, and Trevor I Dix. Compression of strings with approximate repeats. In *ISMB*, pages 8–16, 1998.

[2] Behshad Behzadi and Fabrice Le Fessant. Dna compression challenge revisited: a dynamic programming approach. In *Annual Symposium on Combinatorial Pattern Matching*, pages 190–200. Springer, 2005.

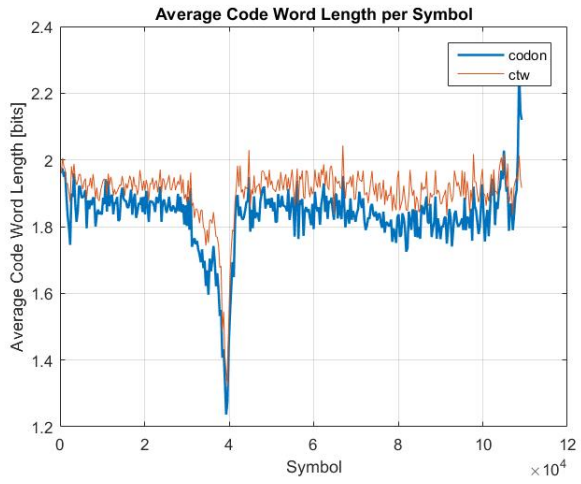

Fig. 5. Graph showing the average code world length of the CTW method against the adapted Codon CTW. Sequence used was ENST00000589042.5

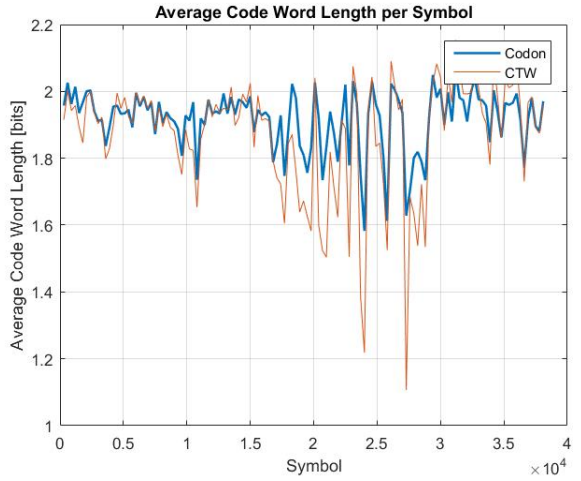

Fig. 6. Graph showing the average code world length of the CTW method against the adapted Codon CTW. Sequence used was NG_027688.1

[3] CL Biji and Achuthsankar Nair. Benchmark dataset for whole genome sequence compression. *IEEE/ACM transactions on computational biology and bioinformatics*, 2016.

[4] Minh Duc Cao, Trevor I Dix, Lloyd Allison, and Chris Mears. A simple statistical algorithm for biological sequence compression. In *Data Compression Conference, 2007. DCC'07*, pages 43–52. IEEE, 2007.

[5] Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for dna sequences and its applications in genome comparison. In *Proceedings of the fourth annual international conference on Computational molecular biology*, page 107. ACM, 2000.

[6] Xin Chen, Ming Li, Bin Ma, and John Tromp. Dnacompress: fast and effective dna sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

[7] Stéphane Grumbach and Fariza Tahi. Compression of dna sequences. In *Data Compression Conference, 1993. DCC'93.*, pages 340–350. IEEE, 1993.

[8] Stéphane Grumbach and Fariza Tahi. A new challenge for compression algorithms: genetic sequences. *Information Processing & Management*, 30(6):875–886, 1994.

[9] Gergely Korodi and Ioan Tabus. An efficient normalized maximum likelihood algorithm for dna sequence compression. *ACM Transactions on Information Systems (TOIS)*, 23(1):3–34, 2005.

[10] Raphail E. Krichevsky and Victor K. Trofimov. The performance of universal encoding. *IEEE Trans. Information Theory*, 27(2):199–206, 1981.

[11] J Kevin Lanctot, Ming Li, and En-hui Yang. Estimating dna sequence entropy. In *Symposium on discrete algorithms: proceedings of the eleventh annual ACM-SIAM symposium on discrete algorithms*, volume 9, pages 409–418, 2000.

[12] David Loewenstern and Peter N Yianilos. Significantly lower entropy estimates for natural dna sequences. *Journal of computational Biology*, 6(1):125–142, 1999.

[13] Giovanni Manzini and Marcella Rastero. A simple and fast dna compressor. *Software: Practice and Experience*, 34(14):1397–1411, 2004.

[14] Armando J Pinho, Paulo JSG Ferreira, António JR Neves, and Carlos AC Bastos. On the representability of complete genomes by multiple competing finite-context (markov) models. *PloS one*, 6(6):e21588, 2011.

[15] Eric Rivals, J-P Delahaye, Max Dauchet, and Olivier Delgrange. A guaranteed compression scheme for repetitive dna sequences. In *Data Compression Conference, 1996. DCC'96. Proceedings*, page 453. IEEE, 1996.

[16] Ioan Tabus, Gergely Korodi, and Jorma Rissanen. Dna sequence compression using the normalized maximum likelihood model for discrete regression. In *Data Compression Conference, 2003. Proceedings. DCC 2003*, pages 253–262. IEEE, 2003.

[17] Frans Willems, Yuri Shtarkov, and Tjalling Tjalkens. Reflections on "the context tree weighting method: Basic properties". *Newsletter of the IEEE Information Theory Society*, 47(1), 1997.

[18] Frans MJ Willems and Tjalling J Tjalkens. Complexity reduction of the context-tree weighting method. In *SYMPOSIUM ON INFORMATION THEORY IN THE BENELUX*, pages 123–130. TECHNISCHE UNIVERSITEIT DELFT, 1997.

[19] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.