

Overview of the Context Tree Weighting version 0.1 implementation

Authors: Erik Franken and Marcel Peeters

Signal Processing Systems group,
Eindhoven University of Technology

More information on CTW and downloads
can be found on the CTW homepage:
<http://www.ele.tue.nl/ctw>

Contents

- 1. Introduction 2
- 2. Probability estimation 4
 - 2.1. Krichevski-Trofimov estimator 4
 - 2.2. Zero-redundancy estimator 5
- 3. Weighting arithmetic 7
 - 3.1. Integer arithmetic 7
 - 3.2. Weighting using quotients of probabilities 7
 - 3.3. Logarithmic representation of probabilities and quotients 9
- 4. Organization of tree-structure 11
 - 4.1. Binary decomposition 11
 - 4.2. Weighting at byte-boundaries 11
- 5. Unique path pruning 13
 - 5.1. Original implementation of unique path pruning 13
 - 5.2. Strict unique path pruning 15
 - 5.3. Handling large files 15
- 6. Hashing 17
- 7. Miscellaneous 19
 - 7.1. Root weighting 19
 - 7.2. Using phase in the hashing function 20
- References 22

1. Introduction

The CTW encoder (and also the decoder) actually consists of two parts (figure 1.1):

- A source modeller (the actual CTW algorithm), which accepts the uncompressed data and estimates the probability on the next symbol.
- An arithmetic encoder, which uses the estimated probabilities to compress the data generated by the actual source. Any arithmetic encoder could be used for this purpose and therefore this part of the encoder will not be discussed in this article.

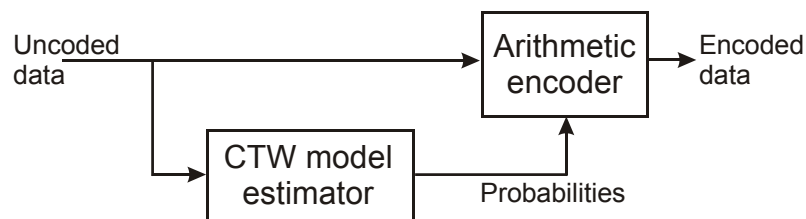


Figure 1.1: CTW encoder

A very important property of the CTW (Context Tree Weighting) algorithm is the context tree, which is built dynamically during the encoding/decoding process:

- The context is defined as all preceding symbols of the symbol that is being encoded. Every context that occurs is stored as a path in the context trees.
- The context trees are 256-ary trees with a certain maximum depth (to limit the amount of memory required). The reason that 256-ary tree structures are used, is that weighting only takes place at byte boundaries ([chapter 4](#)), which is especially suitable for byte-oriented files (e.g. files containing text).
- Each node of a context tree contains two types of data ([figure 5.4](#)):
 1. Four bytes of data required for managing the tree-structure.
 2. Four bytes of data required for estimating and weighting the probabilities, which will eventually be used in the arithmetic encoder and decoder.
- The trees require a lot of memory and therefore need to be stored in an efficient way. In this implementation of CTW, hashing ([chapter 6](#)) is used for this purpose. To limit the number of nodes in a tree, the paths in the tree are not always extended to the maximal depth. This is called unique path pruning ([chapter 5](#)) and results in a great reduction of the required number of nodes (and thus the required amount of memory). Unique path pruning does not affect the compression rate, but it does lead to a small increase in complexity.

When encoding a certain bit, the following steps are performed:

- The path in the context tree that coincides with the current context is searched and, if needed, extended (to make sure all paths are unique, see [chapter 5](#)). This means the symbols from the context are used to decide in which direction the path continues from every node.
- In every node on this context path, the estimated probability on the next symbol P_e is calculated, using the data that is stored inside the node. This estimated probability is calculated with the Krichevski-Trofimov or Zero-Redundancy estimator ([chapter 2](#)).
- Then, a weighted probability P_w is calculated using a weighting function on all of the P_e values ([chapter 3](#)). The idea behind this is that if good coding distributions for two

different sources are weighted, then the weighted distribution is a good coding distribution for both sources (see [IEEE par. 9]).

- Finally the weighted probability is sent to the arithmetic encoder, which encodes the symbol, and then the tree is updated with the new data.

A “mini-course” on the basic properties of the CTW algorithm can be found in [IEEE], which is considered to be understood before reading this article. The main program file is *ctw.c* and the actual steps required for encoding and decoding can be found in *ctwencdec.c*.

2. Probability estimation

In every node along the path in a context tree, the probability on the next symbol is estimated based on the counts of zeros and ones in this node. This is an estimation of the probability that a memoryless source (with an unknown parameter θ) generates X_t as its next symbol. In this implementation of CTW, there are two different estimators to choose from: the Krichevski-Trofimov (KT) estimator and the zero-redundancy estimator (default).

2.1. Krichevski-Trofimov estimator

To estimate the probability on the next symbol for a memoryless source with an unknown parameter θ (i.e. the probability of generating a 1), the Krichevski-Trofimov estimator can be used. The KT-distribution is defined by the following conditional probability:

$$P_e(X_t = 1 | x_1^{t-1}) = 1 - P_e(X_t = 0 | x_1^{t-1}) = \frac{b + \frac{1}{2}}{a + b + 1}, \quad (2.1)$$

where a is the number of zeroes and b is the number of ones in the sequence x_1^{t-1} , which was generated by this source before. In this CTW implementation, the estimated conditional probabilities are represented by their logarithms for easier multiplying and dividing (see [chapter 3](#) about weighting arithmetic). The logarithmic representations of the estimated conditional probabilities are (with all logarithms base 2):

$$\log P_e(X_t = 0 | x_1^{t-1}) = \log \frac{a + \frac{1}{2}}{a + b + 1} = \log(2a + 1) - 1 - \log(a + b + 1) \quad (2.2)$$

$$\log P_e(X_t = 1 | x_1^{t-1}) = \log \frac{b + \frac{1}{2}}{a + b + 1} = \log(2b + 1) - 1 - \log(a + b + 1) \quad (2.3)$$

In the actual implementation, these logarithms can be found in a log-table, which is explained in [chapter 3](#) about weighting arithmetic.

To analyze how well this estimator performs; the “parameter redundancy” can be calculated, which is the part of the total redundancy that results from not knowing the parameter θ of the memoryless source. This redundancy can be calculated by first deriving an expression for the estimated block probability $P_e(a, b)$, which is the estimated probability of a certain sequence with a ones and b zeroes. For $a + b \geq 1$, this estimated block probability satisfies:

$$P_e(a, b) \geq \frac{1}{2} \cdot \frac{1}{\sqrt{a + b}} \left(\frac{a}{a + b} \right)^a \left(\frac{b}{a + b} \right)^b, \quad (2.4)$$

which is proved in [EIDMA ch. 3 par. 10]. Using this lowerbound on $P_e(a, b)$, the parameter redundancy can be bounded by:

$$\rho(x_1^T) = \log \frac{(1-\theta)^a \theta^b}{P_e(a,b)} \leq \log \frac{(1-\theta)^a \theta^b}{\frac{1}{2} \frac{1}{\sqrt{a+b}} \left(\frac{a}{a+b}\right)^a \left(\frac{b}{a+b}\right)^b} \leq \frac{1}{2} \log(a+b) + 1 \quad (2.5)$$

When a source generates zeroes or ones only, the sequences generated by this source do not contain any information and therefore encoding such a sequence wouldn't take any bits in the ideal case. However, the parameter redundancy introduced by the KT-estimator (equation 2.5) grows logarithmic with the number of zeroes (or ones) in this case. This means the parameter redundancy gets quite large if the number of generated zeroes (or ones) gets large.

2.2. Zero-redundancy estimator

Instead of the KT-estimator, the zero-redundancy (ZR) estimator could be used to reduce the parameter redundancy for a source that generates zeroes or ones only. The ZR-estimator is defined as:

$$P_e^{zr}(a,b) = \begin{cases} \frac{1}{2} P_e(a,b) & \text{for } a > 0, b > 0, \\ \frac{1}{2} P_e(a,0) + \frac{1}{4} & \text{for } a > 0, b = 0, \\ \frac{1}{2} P_e(0,b) + \frac{1}{4} & \text{for } a = 0, b > 0, \text{ and} \\ 1 & \text{for } a = b = 0, \end{cases} \quad (2.6)$$

where $P_e(a,b)$ are the block probabilities estimated by the KT-estimator. Now, the parameter redundancy for a sequence generated by a source that generates zeroes or ones only, can be bounded by:

$$\rho(x_1^T) = \log \frac{(1-\theta)^a \theta^b}{P_e^{zr}(a,b)} \leq \log \frac{1}{\frac{1}{4}} = 2 \quad (2.7)$$

This means that the parameter redundancy for such a source is never larger than two bits (for any number of zeroes or ones), which is far better than for the KT-estimator. The parameter redundancy for a source which generates zeroes as well as ones, can now be bounded by:

$$\rho(x_1^T) = \log \frac{(1-\theta)^a \theta^b}{P_e^{zr}(a,b)} \leq \frac{(1-\theta)^a \theta^b}{\frac{1}{2} P_e(a,b)} \leq \frac{1}{2} \log(a+b) + 2 \quad (2.8)$$

So for a sequence containing zeroes as well as ones, the parameter redundancy is only one bit larger than for the KT-estimator (equation 2.5). Since a lot of byte-values never occur in normal text, many of the estimators in CTW only estimate probabilities on sequences which contain zeroes or ones only. Therefore the zero-redundancy estimator usually gives a better result than the KT-estimator, because of these non-occurring symbols.

This CTW implementation uses conditional probabilities. The conditional probability on the next symbol for $a > 0$ and $b > 0$ (i.e. zeroes as well as ones have been generated until now), is the same as for the Krichevski-Trofimov estimator. If only zeroes have been generated so far, the conditional probability that the next symbol will be "one" can be written as:

$$P_e^{zr}(1|0^a) = \frac{\frac{1}{2}P_e(a,1)}{\frac{1}{2}P_e(a,0) + \frac{1}{4}} = \frac{\frac{1}{2}P_e(a,0)\frac{1}{a+1}}{\frac{1}{2}P_e(a,0) + \frac{1}{4}} = \frac{P_e(a,0)}{2(a+1)P_e(a,0) + a + 1} \quad (2.9)$$

Similarly, the conditional probability that the next symbol will be “zero” is:

$$P_e^{zr}(0|0^a) = 1 - P_e^{zr}(1|0^a) = \frac{(2a+1)P_e(a,0) + a + 1}{2(a+1)P_e(a,0) + a + 1} \quad (2.10)$$

If only ones have been generated so far, the same equations hold (of course with the 0’s and 1’s and the a ’s and b ’s interchanged). These conditional probabilities will also be represented by their logarithms in this implementation. To be able to calculate these conditional probabilities, the estimated block probabilities from the Krichevski-Trofimov estimator $P_e(a,0)$ are required. These can be calculated sequentially (for $a = 1, 2, \dots$), using:

$$P_e(a+1,0) = P_e(a,0) \frac{a + \frac{1}{2}}{a + 1} \quad (2.11)$$

The same goes for $P_e(0,b)$ for $b = 1, 2, \dots$. In the actual implementation the conditional probabilities (from equations 2.9 and 2.10) are stored in two tables for all possible values of a and b , which is further explained in [chapter 3](#) about weighting arithmetic.

The probability estimators are implemented in *ctwmath.c*. The log-tables that are used for these estimations can be found in *ctwmath-tables.h* and these are generated using *ctw_gentables.c*. More information on the Krichevski-Trofimov estimator can be found in [EIDMA ch. 3 par. 5] and [EIDMA ch. 3 par. 10]. More information on the zero-redundancy estimator can be found in [EIDMA ch. 5 par. 5].

3. Weighting arithmetic

Limiting the number of arithmetic operations that is required for the CTW algorithm is important to speed up the encoding/decoding process. That's why there are several principles used to make the algorithm faster and more efficient:

- Using integer arithmetic;
- Weighting using quotients of probabilities;
- Logarithmic representation of probabilities and quotients.

3.1. Integer arithmetic

This implementation of the CTW algorithm uses integer arithmetic, because this has two benefits:

- in general integer arithmetic is much faster than floating point arithmetic;
- integer calculations should give exactly the same results on different processor architectures, while floating point calculations can give slightly different results. These differences occur because each processor architecture uses its own floating point representation and its own way to handle the arithmetic operations. This may cause differences in rounding errors.

When using probabilities and logarithms, like in CTW, just truncating all values to integer numbers is not accurate enough. To get enough accuracy while only storing integer values, it is necessary to multiply all values with a factor. This factor is called *ACCURACY* in the source code.

3.2. Weighting using quotients of probabilities

In [IEEE par. 10.2] a formula is derived that can be used to calculate a weighted probability in a node, using the number of ones and zeroes that occurred so far (in this node) and the weighted probabilities from all child nodes. This basic formula for weighting in an internal tree node is:

$$P_w^s(x_1^{t-1}, X_t = 0 | x_{1-D}^0) = \frac{P_e(a_s, b_s) + \prod_{\varphi} P_w^{\varphi s}}{2}, \quad (3.1)$$

where s represents the current context, x_1^{t-1} represents the subsequence from the complete source sequence with all symbols that have context s , X_t is the current symbol, x_{1-D}^0 represents the initial context with depth D (i.e. the first D symbols of the complete source sequence), and $P_w^{\varphi s}$ represents the weighted probability for the (child) node with context φs (with $\varphi = 0$ or $\varphi = 1$). It will be shown that it is a good idea to introduce two quotients of certain probabilities. First, the quotient η is a parameter that is emitted by each node to its parent node on the context path:

$$\eta^s(x_1^{t-1} | x_{1-D}^0) = \frac{P_w^s(X_t = 0 | x_1^{t-1}, x_{1-D}^0)}{P_w^s(X_t = 1 | x_1^{t-1}, x_{1-D}^0)} \quad (3.2)$$

Note that in the leafs of the tree, η is calculated using the estimated probabilities (P_e) instead of the weighted probabilities (P_w), because no weighting can be performed in leafs.

Second, each internal node contains a quotient β which is defined as follows:

$$\beta^s(x_1^{t-1} | x_{1-D}^0) = \frac{P_e^s(x_1^{t-1} | x_{1-D}^0)}{\prod_{\varphi} P_w^{\varphi s}(x_1^{t-1} | x_{1-D}^0)} \quad (3.3)$$

The way these quotients are used is schematically shown in figure 3.1. It can be seen that each internal node contains a quotient β and the quotient η is emitted from child node to parent node on the current context path. Note that the leafs in figure 3.1 don't contain a β , because it is not needed there. However in the actual implementation the leafs have the same data structure, which means all leafs contain a β which always contains the value 1.

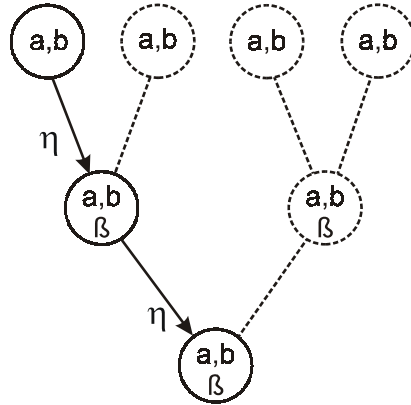


Figure 3.1: the use of quotients η and β . The current context path is indicated by the solid arrows.

Using the quotients η and β has several benefits:

- Less memory space is required: only a , b and β have to be stored in each internal node, instead of memory consuming probabilities as in earlier CTW implementations [EIDMA ch. 5 par. 1]. Using 8 bits for a and b , and between 10 and 16 bits for β appeared to be a good choice.
- Only nodes on the current context path have to be considered. In the basic weighting equation 3.1 it can be seen that the weighted probabilities of all child nodes are required, not only the child node on the context path. In the method described here, this isn't needed anymore because they are implicitly stored in the parameter β^s during earlier 'visits' to the node.

As can be seen in equation 3.6, β^s is some kind of switch that controls the mixture between the incoming probabilities and the probabilities estimated in this node. In our implementation, β^s can be bounded to a certain maximum value, which makes it possible to bound the amount of mixture. Changing the maximum value of beta can affect the compression rate. For example a smaller value can result in a better

compression rate for non-stationary data types, because with a large value of β^s it takes a longer time for β^s to adapt itself to the changing characteristics of the source.

- All calculations can easily be made logarithmic. This has several advantages, which are described in paragraph 3.3.

Weighting in an internal tree node using the quotients η and β requires the following steps:

- The quotient $\eta^{\text{qs}}(x_1^{t-1} | x_{1-D}^0)$ enters node s . The child of s in the current context emitted this quotient.
- Using η^{qs} the weighted probabilities from the child node are calculated:

$$P_w^{\text{qs}}(X_t = 0 | x_1^{t-1}, x_{1-D}^0) = \frac{\eta^{\text{qs}}}{1 + \eta^{\text{qs}}} \quad \text{and} \quad P_w^{\text{qs}}(X_t = 1 | x_1^{t-1}, x_{1-D}^0) = \frac{1}{1 + \eta^{\text{qs}}} \quad (3.4)$$

- Using a_s and b_s (the counts that are stored in node s), the estimated probabilities are calculated, using the KT estimator or ZR estimator. (If the ZR estimator is used, the formulas are slightly different, this is explained in [chapter 2](#) about probability estimation):

$$P_e^s(X_t = 0 | x_1^{t-1}, x_{1-D}^0) = \frac{a_s + 1/2}{a_s + b_s + 1} \quad \text{and} \quad P_e^s(X_t = 1 | x_1^{t-1}, x_{1-D}^0) = \frac{b_s + 1/2}{a_s + b_s + 1} \quad (3.5)$$

- Using these probabilities and the internal quotient β^s , the outgoing quotient η^s is calculated:

$$\eta^s(x_1^{t-1} | x_{1-D}^0) = \frac{\beta^s(x_1^{t-1} | x_{1-D}^0)P_e^s(X_t = 0 | x_1^{t-1}, x_{1-D}^0) + P_w^{\text{qs}}(X_t = 0 | x_1^{t-1}, x_{1-D}^0)}{\beta^s(x_1^{t-1} | x_{1-D}^0)P_e^s(X_t = 1 | x_1^{t-1}, x_{1-D}^0) + P_w^{\text{qs}}(X_t = 1 | x_1^{t-1}, x_{1-D}^0)} \quad (3.6)$$

- Then, the quotient β^s is updated:

$$\beta^s(x_1^{t-1}, x_t | x_{1-D}^0) = \beta^s(x_1^{t-1} | x_{1-D}^0) \cdot \frac{P_e^s(X_t = x_t | x_1^{t-1}, x_{1-D}^0)}{P_w^{\text{qs}}(X_t = x_t | x_1^{t-1}, x_{1-D}^0)} \quad (3.7)$$

- Finally the internal parameters a_s and b_s are updated. This means that a_s is incremented by 1 if $x_t = 0$ or b_s is incremented by 1 if $x_t = 1$. If one of the counts becomes larger than 255, both counts are divided by two. This is necessary because the register size of a_s and b_s is 8 bit.

3.3. Logarithmic representation of probabilities and quotients

In our implementation, all probabilities and the quotients η and β are represented logarithmic (with base 2). The most important benefit is that the expensive multiplying and dividing operations become adding and subtracting in the logarithmic domain:

$$\log(p_1 \cdot p_2) = \log(p_1) + \log(p_2) \quad \text{and} \quad \log(p_1/p_2) = \log(p_1) - \log(p_2) \quad (3.8)$$

However, adding in the logarithmic domain is more difficult. This problem can be solved using the jacobian logarithm $\log(1 + 2^x)$:

$$\log(p_1 + p_2) = \log p_1 + \log\left(1 + \frac{p_2}{p_1}\right) = \log p_1 + \log\left(1 + 2^{(\log p_2 - \log p_1)}\right) \quad (3.9)$$

All formulas described in paragraph 3.2 can easily be made logarithmic. There are two expensive operations left: the logarithm $\log(x)$ and the jacobian logarithm $\log(1 + 2^x)$. These expensive operations can be eliminated by the use of a logarithmic and a jacobian table. In these tables the values of these functions are stored for a certain range of x . The table values can be found in *ctwmath-tables.h*, which is generated by *ctw_gentables.c*, so these values don't have to be calculated again each time the program is executed.

The logarithmic table has a size of 256 entries by default (defined as *LOGENTRIES* in *ctwmath.h*). The entries are calculated as follows (all logarithms are base 2):

$$CTWlog_i = \lfloor A \cdot \log i + 0.5 \rfloor, \text{ for } 0 \leq i < 512, \quad (3.10)$$

where A refers to *ACCURACY* (defined in *ctwmath.h*), which is 128 by default. The upper bound for the range of i is chosen as 512 because there is no need to calculate logarithms larger than 512 (since always $a + b + 1 < 512$).

The jacobian table has a size of 1152 entries by default (defined as *JACENTRIES* in *ctwmath.h*). The entries are calculated using

$$CTWjac_i = \lfloor A \cdot \log(1 + 2^{i/A}) + 0.5 \rfloor, \text{ for } -1152 < i \leq 0. \quad (3.11)$$

The Jacobian values for $i > 0$ are not needed because p_1 en p_2 in equation 3.9 can be interchanged to make i negative again. The values for $i \leq -1152$ are not needed because they are all equal to zero after rounding.

More information can be found in [IEEE par. 10] and [EIDMA ch. 5 par. 2, 3 and 4]. Weighting arithmetic is implemented in *ctwmath.c*. Calculation of logarithmic and jacobian tables is implemented in *ctw_gentables.c*. The calculated tables can be found in *ctwmath-tables.h*.

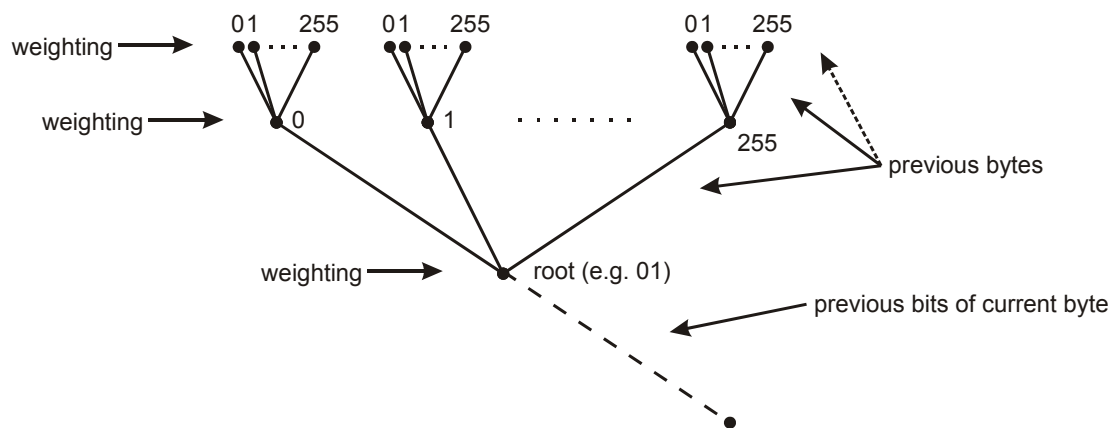


Figure 4.2: weighting takes place at byte boundaries only.

This way the number of nodes in each tree is reduced, which means also the execution time will be lower. For byte-oriented sources this also gives a reduced redundancy, because less bits are required for describing a 256-ary tree than for describing a binary tree (which has more nodes). However, for sources that are not byte-oriented, a 256-ary tree-model might not fit the source as well as a binary tree-model, which might lead to an increased redundancy. Since most sources are byte-oriented, the use of 256-ary trees usually gives a better performance.

The binary decomposition is implemented in *ctwncdec.c*. Weighting is implemented in *ctwmath.c*, so there can be seen that weighting only takes place at byte-boundaries. More information on binary decomposition and weighting at byte-boundaries can be found in [IEEE par. 11].

5. Unique path pruning

Sometimes it is not needed to extend the context tree to the maximum tree depth. If a path in the context tree is a unique path, which means that from a certain level in the tree on the tree doesn't split up anymore. This is because in such a path the weighted probability is equal to the estimated probability in each node. Removing unique pathes from the context tree is called "unique path pruning" (figure 5.1).

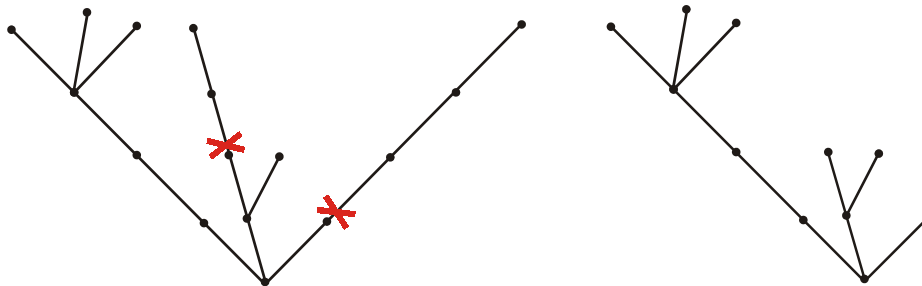


Figure 5.1: with unique path pruning, the tree parts above the crosses (left tree) are removed (right tree)

At a later time during the encoding or decoding process, a context may occur that coincides with a pruned path. Then it is necessary to be able to reconstruct the full context corresponding to this pruned path. If this context is known, it can be compared to the current context, and the tree can be extended as far as needed to make the contexts unique again. However to be able to get the context of a pruned path, it is necessary to keep the whole source sequence in memory, which is a drawback of using unique path pruning. A benefit of using unique path pruning is that the number of nodes decreases significantly, which saves more memory space than the source sequence takes. It can also result in a smaller execution time, because there are less nodes to search for and less calculations to make.

5.1. Original implementation of unique path pruning

In the implementation of CTW as described in [EIDMA], the unique path pruning was implemented as follows:

1. Start at the root node of the context tree for the current phase;
2. find the child node for the current context symbol;
3. if the child node does exist:
 - 3a. the child node becomes the current node;
 - 3b. if the maximum tree depth is reached: stop;
 - 3c. otherwise go back to step 2 to find the next child node;
4. otherwise: (the tree was pruned, going to extend the tree as far as needed)
 - 4a. create the new node;
 - 4b. if the current symbol of the current context is the same as the current symbol of the pruned context: go back to step 4a (to create the next new node of the current context); or if the maximum tree depth is reached: stop;
 - 4c. otherwise: create a node for the pruned context if it doesn't exist yet, and stop.

Example 5.1: Suppose that a tree at a certain time looks like the left tree in figure 5.2.

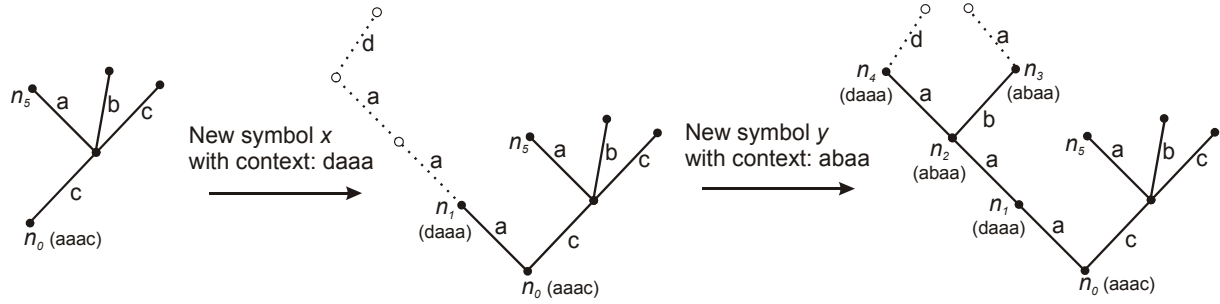


Figure 5.2: pruning example 1

A symbol x with context ‘daaa’ arrives. Note that the context has to be read from right to left, because the symbol on the right side of the total context is the most recent context symbol and therefore the first context symbol to process. The path is searched and updated using the algorithm described above. We start at root node n_0 , which has a pruned context ‘aac’ (i.e. it was previously pruned with this context starting from node n_5), and find the child node for the current context symbol ‘a’. This node does not exist, so we create the node (n_1). The current symbol of the current context ‘a’ is not the same as the current symbol of the pruned context ‘c’, so now we can stop. It is not needed to create a node for the pruned context because it already exists. The dotted line shows the part of the tree that was pruned during this step.

After some time the symbol y with context ‘abaa’ arrives. Again we start at node n_0 . The child node n_1 is found. From node n_1 we search for the child node with context symbol ‘a’. This node does not exist so it is created (n_2). The current symbol of the current context ‘a’ is the same as the current symbol of the pruned context ‘a’, so we create the child node for the current context (n_3). Now the current context symbol of the current context ‘b’ and the one of the pruned context ‘a’ aren’t the same, so the node n_4 for the pruned context is created. The resulting tree is shown on the right side of figure 5.2. Again the dotted lines show the part of the tree that was pruned during this step.

A drawback of this implementation occurs when the new context is exactly the same as the pruned context. In this case it can be seen that the context is fully extended, while this is not necessary at all. This is shown in example 2.

Example 5.2: Suppose that the context tree looks like the left tree in figure 5.3.

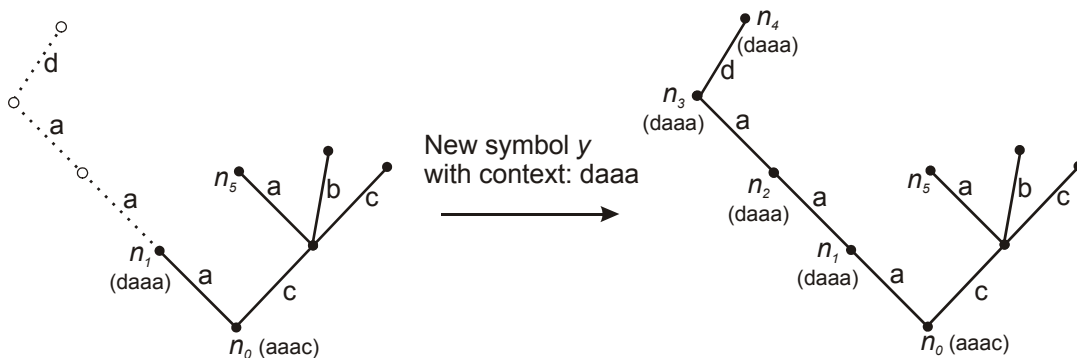


Figure 5.3: pruning example 2

A new symbol y arrives with context ‘daaa’. We start at node n_0 . The child node n_1 is found. From node n_1 we search for the child node with context symbol ‘a’. This node does not exist so it is created (n_2). The current symbol of the current context ‘a’ is the same as the current symbol of the pruned context ‘a’, so we create the child node for the current context (n_3). The next context symbols are also the same (‘d’), so after this node n_4 is created. Now the maximum depth is reached so we stop. The result is that the path for context ‘daaa’ is fully extended, though this is not necessary at all.

5.2. Strict unique path pruning

Strict unique path pruning works a bit different than the original unique path pruning. Before extending the tree, the full context of the pruned tree and the current context are compared with each other. If they are fully the same, no new nodes will be created at all:

1. Start at the root node of the context tree for the current phase;
2. find the child node for the current context symbol;
3. if the child node does exist:
 - 3a. the child node becomes the current node;
 - 3b. if the maximum tree depth is reached: stop;
 - 3c. otherwise go back to step 2 to find the next child node;
4. otherwise: (the tree was pruned, going to extend the tree as far as needed)
 - 4a. *if full context of the pruned tree and full current context are the same: stop;*
 - 4b. (otherwise) create the new node (*copy a and b counts from the last node in the path that already existed to the new node*);
 - 4c. if the current symbol of the current context is the same as the current symbol of the pruned context: go back to step 4b (to create the next new node of the current context); or if the maximum tree depth is reached: stop;
 - 4d. otherwise: create a node for the pruned context if it doesn't exist yet, and stop.

Example 5.3: Consider the left tree in figure 5.3. A new symbol y arrives with context 'daaa'. We start at node n_0 . The child node n_1 is found. From node n_1 we search for the child node with context symbol 'a'. This node does not exist. Now we check if the full context of the pruned tree and the full current context are the same. This is true because they are both 'daaa', so we stop. The tree structure does not change.

The use of strict unique path pruning results in about 15% less nodes (this was the average saving on all files of the calgary corpus) compared to the original unique path pruning algorithm. It also results in a slightly faster execution time, because on average there are less nodes that have to be treated per encoding step, which saves time that was needed for the hashing method to find the node ([chapter 6](#)) and the weighting in the node ([chapter 3](#)). The time that is saved this way appeared to be more than the time that is needed to compare the full contexts, which has to be performed for strict unique path pruning. Another benefit is that because less nodes are created with strict unique path pruning enabled, there usually will be less failed nodes, which results in a better compression performance. However, in theory the compression rate would have to be exactly the same, as long as there are no failed nodes. In practice the number of code bits might differ a little, because of rounding errors in the integer arithmetic ([paragraph 3.1](#)).

5.3. Handling large files

For unique path pruning we need to store the whole file in memory. This leads to a new problem: if the files are very large there is too much memory space required. This means the amount of available memory leads to a limitation on the file size. But if there is plenty of memory, there still is another restriction: the pointer to the filebuffer in each node in our implementation has a size of 24 bits. This means that a file buffer larger than $2^{24} = 16$

megabytes is useless because we cannot address it. There are several possible solutions to this problem:

- The most trivial solution is choosing a larger pointer to the file buffer and a larger file buffer size. The disadvantage of this solution is that the file buffer and the nodes require more memory. Another disadvantage is that the size of each tree node, which is 8 bytes in this implementation (figure 5.4), has to become larger, e.g. 9 bytes. This can lead to slower memory accesses because it is not a power of 2. For example, on a computer with a 32-bit bus to data memory, 8 bytes is exactly 2 words to access. With 9 bytes, 3 accesses are needed per node.
- When the buffer is full, remove the contents of the buffer and start again with a new context tree. This leads to much extra redundancy because the context tree has to be built again.
- When the buffer is full, “freeze” the tree and the contents of the file buffer. This means no new nodes will be added anymore. The rest of the file will be encoded using the current tree and the rest of the file will not be stored in the file buffer. However the a , b and β parameters in each node will still be updated. If the source characteristic is quite stationary, this won't lead to much additional redundancy.

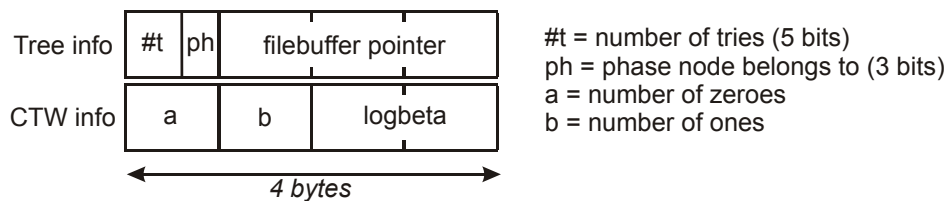


Figure 5.4: structure of a tree node in this implementation

In this implementation we choose the last solution: freezing the tree and file buffer. It is possible to specify the maximum file buffer size on the command prompt. If the file size exceeds this size, the tree will be frozen as soon as the file buffer is full.

More information can be found in [EIDMA ch. 4 par. 1.4]. Unique path pruning is implemented in *ctwtree.c*.

6. Hashing

Storing a tree structure in computer memory can be done in various ways:

- Allocating an array in which the data of all possible nodes in a fully extended tree (to a certain depth) can be stored. The benefit of this method is that a simple calculation can be used to find a child or a parent of a node. For example for a binary tree the children of a node on index x can be found on position $2 \cdot x$ and $2 \cdot x + 1$. However, often a tree (also the context tree of the CTW algorithm) will not be fully extended. This results in a very sparse array and thus a large waste of memory.
- Dynamically allocating memory space for the data of each node. The benefit is that memory space is only required for nodes that are actually created, but the drawback is that extra memory is required to store pointers to the children (and if needed to the parent) of the node. Especially when a lot of nodes have to be allocated with a small amount of data, like in the CTW algorithm, these pointers cause a huge overhead.
- The use of hashing. With hashing, an array is used in which a fixed number of nodes can be placed. The idea is that every node is placed on a pseudo-random index in this array. The index is calculated from a certain *key* parameter with the so called *hashing function*. The hashing function must be a pseudo-random function, but the result of the hashing function must always be the same given the same key value. It is possible that different keys result in the same index; this is called a collision. To detect such a collision it is important that there is some data stored in each node which makes it possible to recognize if the right node has been found. If it isn't the right node, the hashing algorithm can perform a second try, adding a certain offset to the index value and checking if the new location in the array contains the right node. The hashing function must be chosen such that the probability on a collision is as small as possible. The benefit of using hashing is that it's an efficient way of using memory when storing a tree. A drawback is that possibly a node can't be allocated although there still is free space in the array, because of the chosen hashing function and the maximum number of tries that is performed. It is also difficult to forecast how large the array should be.

For use with the CTW algorithm, the hashing method is best suitable. The CTW hashing implementation is simple and fast:

- The index in the tree array of the parent of the node that is to be found, is used as a 'start position', say i .
- A certain offset is calculated with the use of a hashing function, in which a 'mix' of the value of the current context symbol and the position of the current bit in the current context symbol (called *phase*, as described in paragraph 4.1) is used as the key:

$$offset = \text{hash}(symbol) \text{ xor } ((phase+1) \cdot 2), \text{ with } 0 \leq phase \leq 7. \quad (6.1)$$

Because we actually store different trees for each phase in the tree array, the phase is also used in calculating this key because this decreases the probability that two trees conflict with each other. This is explained in more detail in [paragraph 7.2](#). The value of $\text{hash}(symbol)$ is retrieved from the hash table, which is just a permutation table with 256 pseudo-random chosen values.

- A new index is calculated: $i_{new} = i + offset$ (modulo the array size).

- At index i_{new} in the tree array we check if the right node is found. This can be recognized by the *phase* the node belongs to, the *number of tries* that was needed to find this node and the current context *symbol*.
- If all three values are correct, it is sure that the right node is found. If the values are not correct, a new index i_{new} is calculated by adding *offset* to the current index again, and a new try will be performed at this new index. However if the number of tries exceeds the maximum number of tries that is allowed, the search operation will stop. In that case, the node can't be used in the CTW algorithm, which will result in a (slightly) worse compression rate, because now CTW in fact is forced to use a simpler source model. However the file can still correctly be decoded because exactly the same nodes will fail during the decoding process. It is also possible that an empty node is found. If this situation occurs, the node we're searching for is a new one or the tree is pruned ([chapter 5](#)) at that location.

Using hashing appeared to be a good choice. It decreased the amount of required memory enormously. However a drawback still is that it is difficult to estimate how large the array in which the values are stored should be for a certain file. This size mostly depends on the characteristic of the input file and the maximum tree depth. The total array has to be allocated before the actual encoding/decoding process, and at that time the characteristic of the input file is not known yet.

More information can be found in [EIDMA ch. 4 par. 3]. Hashing is implemented in *ctwtree.c*.

7. Miscellaneous

Beside the subjects discussed in the previous chapters, there are some subjects that are not so relevant for understanding the current implementation of CTW. These subjects can be found in the implementation however, so they will be discussed in this chapter. First the effects of root weighting will be discussed (which can be enabled but is disabled by default). In the second paragraph it is explained why the number of the bit in the current byte (the *phase*) is used in the hashing function.

7.1. Root weighting

The part about binary decomposition in [chapter 4](#), states there are 255 different trees. However, one could also see this as if there is just one tree for each *phase* (bit) of an ASCII-symbol, which first splits in 2^{phase} nodes (with $0 \leq \text{phase} \leq 7$) dependent on the previous bits of the current symbol. Now each of these nodes corresponds to the root node of one of the 255 trees mentioned before.

If root weighting is disabled, the previous bits of the current symbol are used to select the weighted probability from one of these nodes only. This means the weighted probability from the selected node is used as the coding probability. However, if root weighting is enabled, weighting also takes place in the root nodes of the eight trees (not to be confused with the root nodes of the 255 trees). For example, the coding probability (i.e. weighted probability in the root node λ) for the third bit of a byte (*phase* = 2) is given by:

$$P_w^\lambda = \frac{P_e^\lambda(x_t) + P_w^{00}(x_t)P_w^{01}(x_t)P_w^{10}(x_t)P_w^{11}(x_t)}{2} \quad (7.1)$$

If the symbol source that is modeled really is a source with a 256-symbol alphabet, then the most reasonable choice is to disable root weighting. If the true source model does not include the root node, the codeword should be about one bit shorter per tree than with root weighting enabled, so the total gain would be about eight bits. For the Calgary corpus, the difference in the codelength per file varies from -42 to $+120$ bits (for depth 4) and from -43 to $+110$ bits (for depth 8), as can be found in [EIDMA ch. 4 app. 4]. Without root weighting the speed is also increased slightly, because less weighting operations are to be performed. However, in the implementation as described in [EIDMA] root weighting was enabled, because enabling root weighting does not much influence the performance of the algorithm and the implementation with root weighting enabled is slightly simpler (because the root nodes are handled in the same way as all other nodes). In this implementation of CTW root weighting can be enabled or disabled using command line options, but is disabled by default. Enabling root weighting might result in a slightly different (better or worse) compression performance, dependent on the characteristics of the input file.

7.2. Using phase in the hashing function

A simple hashing function that could be used in this implementation is:

$$\text{offset} = \text{hash}(\text{symbol}) \quad (7.2)$$

However in chapter 6, we introduced a more difficult hashing function:

$$\text{offset} = \text{hash}(\text{symbol}) \text{ xor } ((\text{phase}+1) \cdot 2) , \text{ with } 0 \leq \text{phase} \leq 7 \quad (7.3)$$

One might wonder what is the benefit of using *phase* in the hashing function. Suppose that we use formula 7.2 as hashing function. Now the offset only depends on *symbol*, which means in all different trees the offset (that is added to the index of a node into the array) for a certain value of *symbol* is the same. This may lead to a situation in which collisions occur between two trees that try to store a node on the same positions all the time. The trees that belong to different phases look quite similar to each other because the context symbols are the same for all phases, so this situation will occur regularly. When using phase, the offsets become different for the same value of symbol in different trees. This results in a smaller probability for a collision to occur, which results in a faster execution time.

The effect of using *phase* can be measured by calculating the average number of tries that was needed to find a node during the encoding / decoding of a file. A smaller average number of tries means a smaller amount of collisions. Table 7.1 shows the results for all files in the calgary corpus.

<i>Filename</i>	<i>No phase</i>	<i>With phase</i>	<i>difference</i>
bib	1.09808	1.09732	-0.00076
book1	1.21630	1.21250	-0.00380
book2	1.19481	1.19256	-0.00225
geo	1.16459	1.16006	-0.00453
news	1.19255	1.19038	-0.00217
obj1	1.05890	1.05101	-0.00789
obj2	1.16199	1.14584	-0.01615
paper1	1.12991	1.12901	-0.00090
paper2	1.15380	1.15307	-0.00073
paper3	1.14326	1.14303	-0.00023
paper4	1.11142	1.11149	-0.00007
paper5	1.10925	1.10833	-0.00092
paper6	1.12573	1.12459	-0.00114
pic	1.09279	1.08770	-0.00509
progc	1.10625	1.10515	-0.00110
progl	1.09582	1.09543	-0.00039
progp	1.07847	1.07805	-0.00042
trans	1.08486	1.08360	-0.00126
<i>average</i>	1.12882	1.12606	-0.00276

Table 7.1: average number of tries for each file in calgary corpus

It can be seen that the average number of tries decreases for all files with the use of *phase* in the hashing function. The differences may seem to be small, but this is because the values are averages on the whole file and in most cases one try will be enough.

Root weighting can be found in *ctwmath.c*. More information on root weighting can be found in [EIDMA ch. 4 par. 4].

References

- [IEEE] F.M.J. Willems, Y.M. Shtarkov and Tj.J. Tjalkens, "Reflections on 'The Context-Tree Weighting Method: Basic Properties'", *Newsletter of the IEEE Information Theory Society*, March 1997
- [EIDMA] F.M.J. Willems and Tj. J. Tjalkens, "Complexity Reduction of the Context-Tree Weighting Algorithm: A Study for KPN Research", EIDMA Report RS.97.01, Technical University of Eindhoven, 1997

The [IEEE] article is a reflection the following article, which might be interesting if you want to know more about the basic properties:

- F.M.J. Willems, Y.M. Shtarkov and Tj.J. Tjalkens, "The Context-Tree Weighting Method: Basic Properties", *IEEE Transactions on Information Theory*, Vol. 41, No. 3, May 1995