PAQ compression algorithm

Krzysztof Blaszczyk
RWTH Aachen University

Supervisors:
Prof. Dr. Peter Rossmanith
Dipl-Inf. Alexander Langer
Dipl-Inf. Felix Reidl

June 11, 2012

Contents

Abstract

PAQ is a flexible state-of-the-art compression algorithm that beats compression ratio records und pushes achievable compression ratios to new frontiers. This seminar paper should provide a general understanding of that interesting and astounding algorithm.

1 Introduction

Since the computer revolution in 1980's data compression has become one of the standard tools in computer science. There are several reasons why compression is important, one of the main reasons being that physical devices can only store limited amounts of data. Another motivation are data transfer and data storage limitations. On the other hand, compression itself can require extensive computation and can significantly delay processing of data. However, if it is used wisely, data compression saves resources and money, helps to overcome limits and increases energy efficiency.

Interestingly, in theory, compression of data sequences where symbols are uniformly distributed should not be possible. For example, if there is a set of all binary strings of length n then there is no bijective function between this set and the set of all binary strings with length smaller than n. Obviously if there is no bijection then the Symbol mapping is not reversible and compression is not possible. In contrast to practice, a file that has a uniform distribution of Symbols but is compressible can be found easily. For instance when the first half of a file consists of zeros and the second half consists of ones, the very statement about that property, together with file size information and some fixed agreement about the value of the central bit (when file size is odd) is sufficient to reconstruct that file. Obviously if the file size is large enough, compression can be achieved.
Another interesting result originates form Kolmogorov's complexity[Kol65]. It states that compression limits are incomputable because the shortest possible description of an object is not computable. So even if data can be compressed in practice, there is no way to tell what the theoretical compression limit is.

There are more ways to view the problem. Algorithmic probability theory for example assumes that the larger a description of an object is, the less likely it is to be the explanation for the structure of that object. Using that and other theories and assumptions Hutter [Hut05] claims, that compression is an AI problem. He suggests that the optimal behavior of a goal seeking agent in an unknown but computable environment is to guess at each step that the environment is probably controlled by one of the shortest programs consistent with all interaction so far.

He develops a powerful parameter-free theory of an optimal reinforcement learning agent, where finding such optimal behavior is done by gradually eliminating Turing machines once they become inconsistent with the progressing history. He emphasizes that the same is done in compression when seeking the shortest Turing machine that generates the given data.

In 2006 Hutter announced a competition with 50,000 Euro of total funding and paid 500 Europa for each 1% compression ratio improvement on a specific standard file he provided. At that time, Matt Mahoney, a data compression specialist from the IBM company had an experimental open source project running that he called "PAQ". His goal was to create an archiver that pushes compression to its (incomputable) limits while ignoring space and time complexity constraints as much as possible. Compression ratio achieved by his version "PAQ8F" on the provided standard file was set as the starting point for the competition. Matt Mahoney received a pre-prize from Hutter for already achieving highest known compression ratios. Driven by competition, PAQ continues to evolve. By 2012 there has been over 45 different implementations of the PAQ algorithm.

So what is so special about PAQ that makes it so successful in achieving highest compression ratios? This paper intends to explain the basic principles underlying the algorithm.

2 PAQ Algorithm

2.1 Recap about arithmetic coding

In arithmetic coding [Nim12] there is a set of Symbols where each Symbol has an assigned probability of occurrence. Interval [0,1) is recursively divided by the probability distribution into distinct subintervals where each interval represents a different distinct symbol. This is done recursively until a terminating symbol occurs or until some predefined maximum recursion depth is reached.

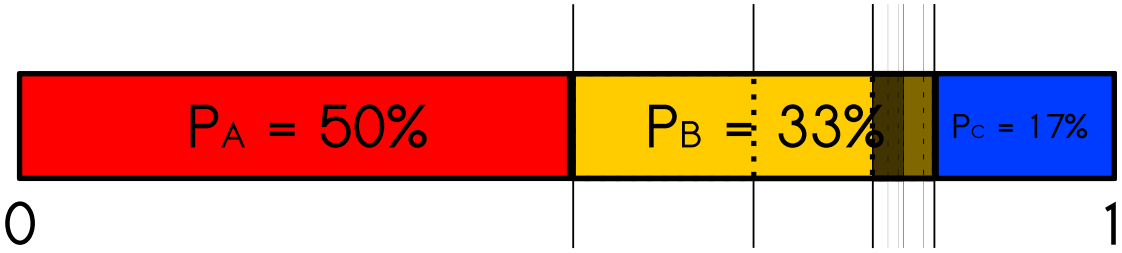

Figure 1: Visualization of arithmetic coding by assuming fixed probability distribution and fixed recursion depth

Consider that in Figure 5 the probability distribution does not have to be fixed. It could be made adaptive and change in each recursion step. As long as the adaptation process is deterministic the compression will be reversible.

2.2 Recap about PPM

PPM [Sew12] is strictly spoken not a compression algorithm, but a prediction algorithm that can be used with other compression algorithms like arithmetic coding. PPM outputs a probability distribution for the next Symbol based on a fixed length data window (also called context) on previously seen data. Obviously future data can not be taken into account, because the decompressor can not see it and therefore could not deterministically reconstruct the sequence of probabilities used. The prediction is then passed to a coder that can exploit such data in attempt to output a more compressed sequence. The most common use of PPM is to use it together with an arithmetic coder since arithmetic coding is known for its optimality.

2.3 Connection to PAQ

PAQ Algorithm is an evolution of PPM and combined with a standard arithmetic coder as well. PAQ mainly differs from PPM in following areas:

- Context becomes an arbitrary complex function of already seen data
- Instead of only one model (predictor) there are many different models

2.3.1 Contexts

Except for taming memory complexity, there is no other obvious reasons to restrict or fix the context size. Instead of limiting which past data can be accessed, maximal possible freedom is given. When predictions can be made more accurate by making more past data usable, then it should be allowed to do so. And because a context is a function of that data, it can be used as a filter or modifier to extract information that is relevant to prediction while ignoring parts that are known for not correlating with future data.

One context example is to select higher order bits only which can be advantageous for prediction in data formats where these bits relate to each other. Another example could be a N-gram which is a selection of n most recently seen bits, A Context could also be some kind of a hash function or even a constant. Only usefulness matters here.

2.3.2 Models

PAQ calls its predictors "models". A model is always connected to a context which defines the input on which the prediction should be based on. Models make predictions by assuming some specific information patterns in what they can see through their contexts. One specific model can occur many times but be connected to different contexts and one specific context can be used many times in different models. A more advanced example could be a model which assumes that values follow some sinus-like pattern and perform prediction by approximating the data with some sinus function. This might be the case in analog data such as audio files. In this light it might be interesting to mention that Jeff Hawkins theorizes in his book "On intelligence"[Haw04] that intelligence is just the ability to predict the future by learning from past data.

2.4 Context Mixing

Arithmetic coding always needs one single prediction at a time. Having many different models connected to many different contexts that output many individual predictions is incompatible with an arithmetic coder at first. A way to combine all the predictions into one single prediction is needed. In addition there is hope that this final, single prediction, which can easily be based on much wider world knowledge, will be more accurate than the single probability distributions on their

own. The "art" of combining these multiple predictions into one single probability distribution is called "context mixing". The word "context" in "context mixing" is not to being confused with the definition of contexts for models. A more accurate name would probably be "prediction mixing" but this unfortunately is not the standard phrase used.

Through time, PAQ evolved the way of performing context mixing to be more and more sophisticated. In this paper three major steps of that evolution will be described.

In all cases, context mixing involves assigning weights to all models used before combining them together. Since PAQ always predicts exactly one bit, the alphabet is $\Sigma_{PAQ} = \{0, 1\}$, where $[\![0]\!] = 0$, $[\![1]\!] = 1$ and the final prediction is always a pair of numbers, one of them representing the probability for the next symbol being 0 and the other representing the probability for the next symbol being 1. In detail, context mixers differ in expressing and managing these weights and in using them to calculate the final prediction. The way probability is expressed may also differ as it will be shown in the following sections.

2.4.1 Mixing by weighted averaging

The very first context mixing method that PAQ used was simple model weighting. Here, each model $m \in \{1, ..., n\}, n \in \mathbb{N},\ n\ is\ number\ of\ models$ has a fixed weight $w_m \in \mathbb{N}$ and the probabilities are expressed as a pair of natural numbers. Normalization of weights or probabilities is not needed, because it is the ratio between the weights that matters. In all following formulas it is implicitly assumed that contexts are associated to models. Reason for such simplification is that formalizing contexts does not change the basic structure of any formula used in this paper and can therefore be left out. Further, a lower case p will stand for the normalized probability $p \in [0, 1]$ and an upper case P will stand for the raw representation of that probability (i.e as natural number).

Let the probability distribution/prediction by a model m be the tuple

$$P_m = (P_m(0), P_m(1)),$$

then the symbol 0 has the probability

$$p_m(0) = \frac{P_m(0)}{P_m(0) + P_m(1)}$$

and the symbol 1 has the probability

$$p_m(1) = \frac{P_m(1)}{P_m(0) + P_m(1)}$$

Calculation of the final probability P_* is simply the weighted sum:

$$P_* = \sum_{m=1}^{n} P_m w_m = (P_*(0), P_*(1))$$

The values of fixed model weights w_m were arbitrarily assigned by programmers. However, it usually meant that if context had length k, the weight was set to k^2 This is based on the argument that greater contexts would deliver more knowledge to models and models using these contexts would be able to predict more accurately.

Mixing by weight averaging is a relatively primitive form of context mixing. The argument mentioned above that more input means better prediction must not be necessary true. Above all, when using fixed weights some local data patterns might have been predicted much better with different weight values. This inflexibility was later solved as described in following chapters.

2.4.2 Mixing by adaptive model weighting

The obvious evolution of context mixing by weighted averaging is to make the weights w_m adaptive in order to gain local flexibility. The weights must change deterministically in each step to provide a deterministic sequence of predictions. As before, the weights and probabilities are expressed as natural numbers and the calculation of P_* remains the same. The only change is that weights are dynamic now. A question one might raise at this point is how to change the weights in favor of better prediction.

The idea here was making the assumption that "collective wisdom" must usually be greater than "individual wisdom". In other words, one assumed that the final weighted prediction will be more accurate than most of its single predictions. Therefore, the greater the prediction error and the greater the deviation between P_* and P_m, the more adjustment should be done to w_m. Let us define the weighted sum off all 0-symbol predictions and the weighted sum off all 1-symbol predictions as:

$$\Sigma^{0*} := P*(0) = \sum_{m=1}^{n} P_m(0) w_m$$

$$\Sigma^{1*} := P*(1) = \sum_{m=1}^{n} P_m(1) w_m$$

Furthermore:

$$\Sigma^* := \Sigma^{0*} + \Sigma^{1*}$$

Let x be the Interpretation of the actual symbol that a model was attempting to predict:

$$error := x - \frac{\Sigma^{1*}}{\Sigma^*} = x - p_*(1)$$

Note that the error can be negative or positive.

If the error is ***positive*** then $P_*(1)$ should ***increase towards 1***. This can be done by weakening weights of models that predicted the opposite and strengthening weights of models that predicted towards the correct value.

If the error is ***negative*** then $P_*(1)$ should ***decrease towards 0***. This, again, can be done by weakening weights of models that predicted the opposite and strengthening weights of models that predicted towards the correct value.

The next task is to express the "collective wisdom". In an optimal case, a model m is just as accurate as the final prediction. This can be expressed as:

$$\frac{\Sigma^{0*}}{\Sigma^{1*}} = \frac{P_m(0)}{P_m(1)}$$

By applying simple algebra it follows:

$$\Sigma^{0*} P_m(1) - \Sigma^{1*} P_m(0) = 0$$

If P_* and P_m differ only slightly then $d \neq 0$ when d is defined as:

$$d := \Sigma^{0*} P_m(1) - \Sigma^{1*} P_m(0)$$

If d is ***positive*** then

$$\frac{P_m(0)}{P_m(1)} < \frac{\Sigma^{0*}}{\Sigma^{1*}}$$

To make left and right hand side equal again the value of $\frac{P_m(0)}{P_m(1)}$ should ***increase***. This can be done by ***increasing*** the numerator $P_m(0)$ and/or ***decreasing*** the

denominator $P_m(1)$. Therefore if d is positive then, compared to the final prediction, the probability for $x = 1$ is ***too high*** and the probability for $x = 0$ is ***too low***. Without loss of generality it holds:

$$\frac{p_m(0)}{1 - p_m(0)} < \frac{p_*(0)}{1 - p_*(0)}$$

By using simple algebraic transformations it follows that $p_m(0) < p_*(0)$ which also implies $p_m(1) > p_*(1)$

If d is ***negative***, then the reasoning is precisely vice versa.

The weight adjustment is done as follows:

$$w_m = w_m + error \times d$$

Using these findings the intention of the formulas can be shown easily for all 4 cases:

Case 1 : $x = 0$ but $P_*(1) > P_*(0)$ (prediction towards 1 instead of 0)

Case 2 : $x = 1$ but $P_*(1) < P_*(0)$ (prediction towards 0 instead of 1)

Case 3 : $x = 1$ and $P_*(1) > P_*(0)$ (correct prediction towards 1)

Case 4 : $x = 0$ and $P_*(1) < P_*(0)$ (correct prediction towards 0)

Cases 1 and 2 as well as cases 3 and 4 are reversions of each other. To avoid unnecessary redundancies only case 1 and 3 will be analyzed:

Case 1: $x = 0$ **but** $P_*(1) > P_*(0)$

If $x = 0$ but $P_*(1) > P_*(0)$ then $P_*(1)$ should decrease in favor of $P_*(0)$. By the above the error will be ***negative***.
If under these conditions a model m outputs a prediction with $p_m(0) < p_*(0)$ then d will be ***positive***. As result the term $error \times d$ will be ***negative*** and the weight of the model will ***decrease***. This is consistent with making a model which predicts stronger towards 1 less significant.
If under these conditions a model m outputs a prediction with $p_m(0) > p_*(0)$ then d will be ***negative***. As result the term $error \times d$ will be ***positive*** and the weight of the model will ***increase***. This is consistent with making a model which predicts stronger towards 0 more significant.

Case 3: $x = 1$ **and** $P_*(1) > P_*(0)$

If $x = 1$ and $P_*(1) > P_*(0)$ then the goal should be to further increase $P_*(1)$ and decrease $P_*(0)$. By the above the error will be ***positive***.
If under these conditions a model m outputs a prediction with $p_m(1) > p_*(1)$ then d will be ***positive***. As result the term $error \times d$ will be ***positive*** and the weight of the model will ***increase***. This is consistent with making a model which predicts stronger towards 1 more significant.
If under these conditions a model m outputs a prediction with $p_m(1) < p_*(1)$ then d will be ***negative***. As result the term $error \times d$ will be ***negative*** and the weight of the model will ***decrease***. This is consistent with making a model which predicts stronger towards 0 less significant.

One drawback of this method is that all weight adjustments use the same *error*-value. Being able to calculate *error*-values individually for each model and adjust the weights in a more differentiated manner might lead to more flexible ways of context mixing, opening possibilities for further improvement.

2.4.3 Mixing by neural networks

If one examines the previous weight adjustment formula $w_m = w_m + error \times d$ and possesses some knowledge about AI-algorithms and particularly neural networks it probably reminds him of another very similar formula used there:

$$w_m = w_m + error_m \times \alpha \times Input_i [\text{Lak12}]$$

This and the fact that $error_m$ is "some kind of" individual value (explanation follows) immediately suggest to use neural network algorithms to perform context mixing.

2.4.3.1 Neural Networks Neural networks try to imitate information processing as it is done between biological neurons. As shown in figure 2, a neural network is usually a weighted directed graph where nodes represent neurons and weights represent the connection strength between them. There is a layer of input neurons where a signal is "fired" and a layer of output neurons where the signal arrives at some strength. The final strength value represents some result. Signals can only travel in one direction which is the direction towards the output layer. They get distributed over all edges and are multiplied by the weight of the edge they pass.

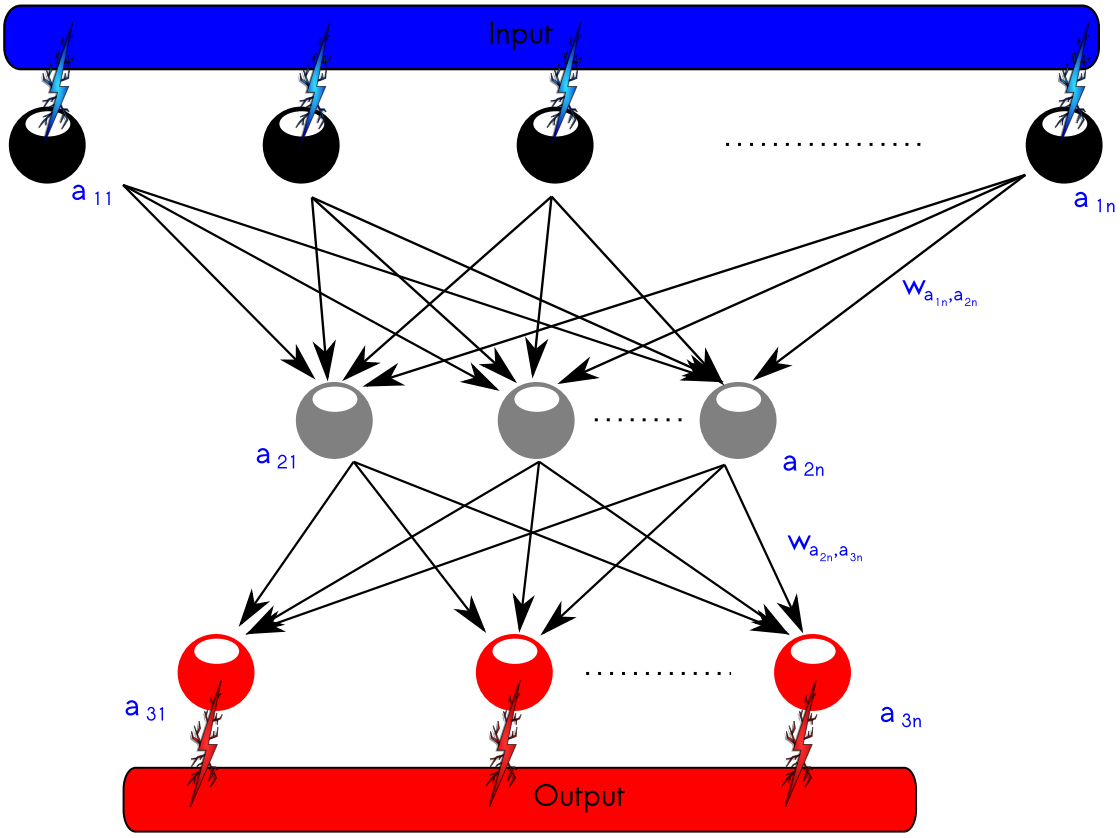

Figure 2: Visualization of a backpropagation neural network with one input layer, one hidden Layer and one output Layer

Weights and signals are usually real numbers between 0 and 1. The incoming signal strength $Input_i$ of a node is a function of the sum of all incoming signals. This function, also called the activation function. It usually normalizes and propagates the newly calculated signal to all neurons in front of it. This continues until the output layer is reached. The interpretation what a strong or weak input/output signal means can be chosen freely which makes the algorithm very flexible. In neural networks, the weights are interpreted as the knowledge of the network and they can be optimized by repetitive firing of different signals with different ideal outputs. This repetitive process of weight adaptation is called "learning" or "training" and one such iteration is called an "epoch". Finally, the layer between the input and the output (so called "hidden layer") can be ignored here, since it is not used in classic PAQ versions.

2.4.3.2 Backpropagation Neural Networks Backpropagation networks are commonly used if the ideal output signal is known. When ideal output signals are

known then for each input node an $error_m$-signal can be calculated and used to guide the learning process. This is done by tracing the signals backwards from Output to Input and adjusting the weights by using individual error values. Historically, it took a very long time to figure out the procedure for propagating the errors back in order to adjust the weights. The official breakthrough happened in 1985[RHW85] while backpropagation networks were already proposed by Alexander Bain in 1873[Bai73].

2.4.3.3 Neural Networks in PAQ In PAQ, the input nodes/neurons can be directly interpreted as models, and the signal weights can be directly interpreted as model weights. Since no hidden layers are used, the network is a complete, directed, weighted bipartite graph. The input signals are functions of $P_m(1)$ for all m and the output signal comes from only one neuron representing $P_*(1)$. Probabilities $P_m(0)$ and the probability $P_*(0)$ can be ignored because they can be deduced by $P_m(0) = 1 - P_m(1)$ and $P_*(0) = 1 - P_*(1)$.

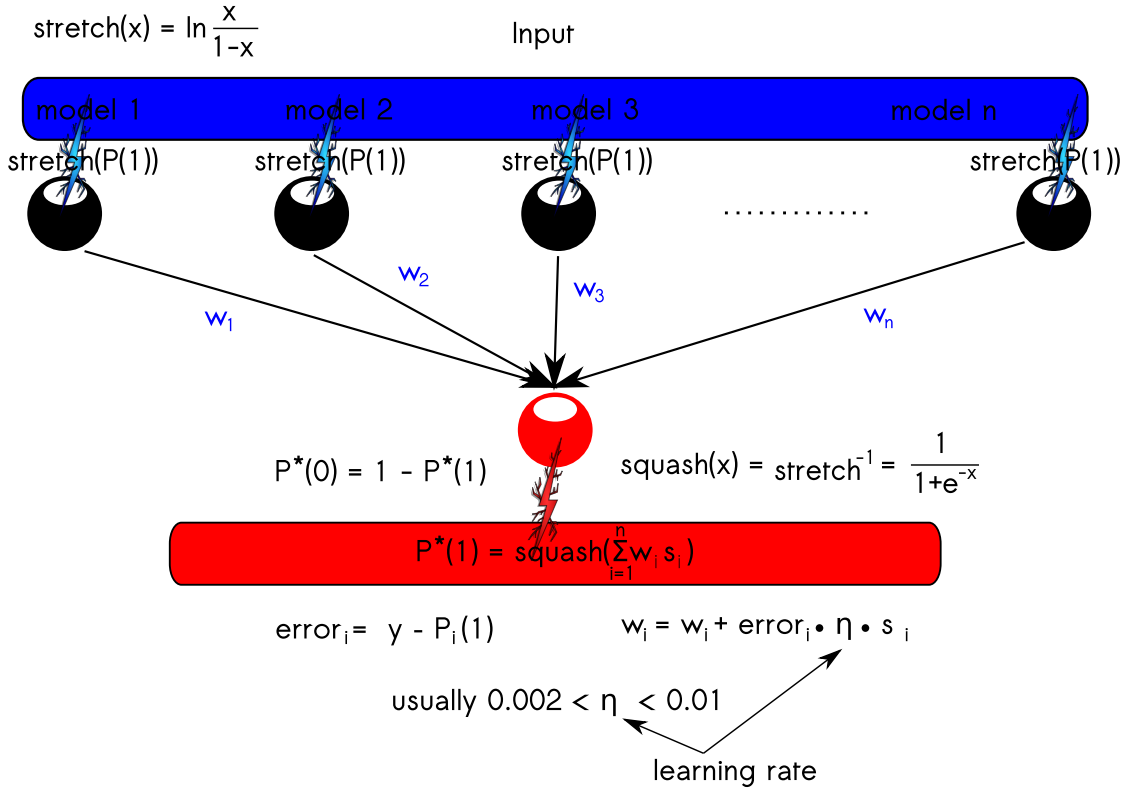

Figure 3: Weight adjustment by using a backpropagation neural network variant.

The function that translates the probabilities $P_m(1)$ into a signal s_m is called

stretch:

$$stretch(x) = logit(x) = ln(\frac{x}{1-x})4$$

In this regard the signal strength s_m that is fired by a particular model m is defined as:

$$s_m := stretch(P_m(1))$$

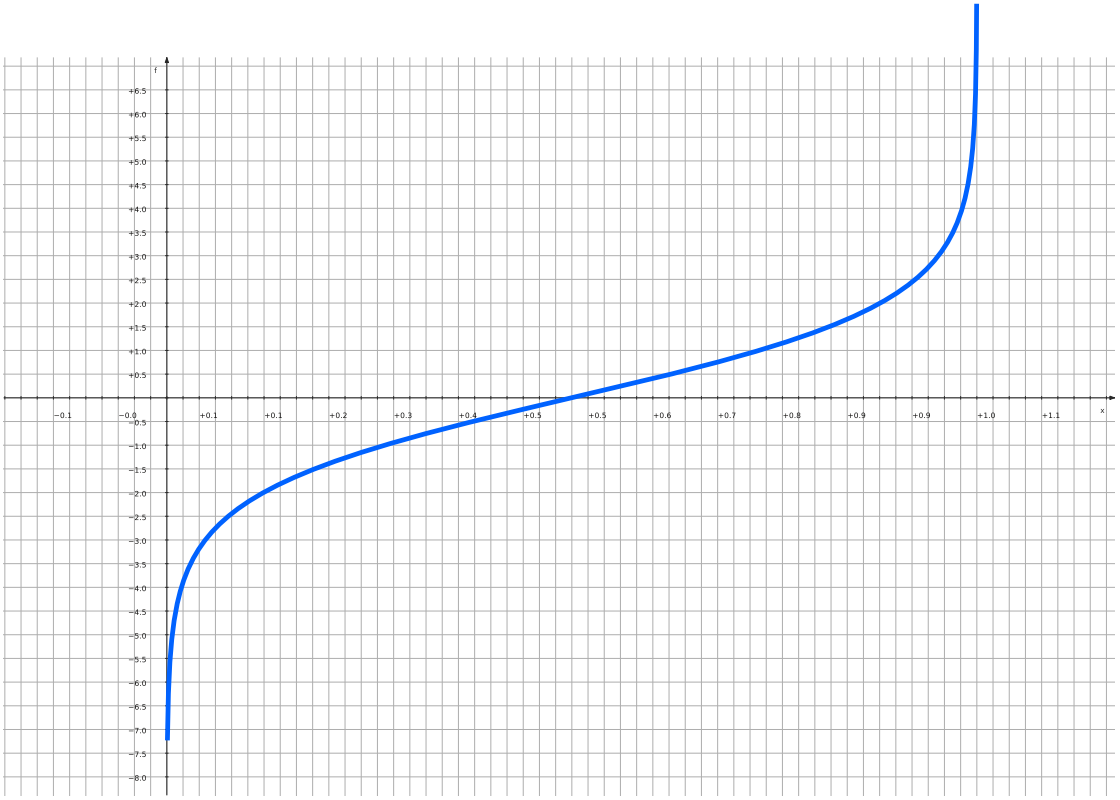

Figure 4: Plot of the logit (stretch) -function.

Notice that if the probability is 0.5, then $stretch(0.5) = 0$ which means there is no signal at all if neither 0 nor 1 can be predicted 4. The output signal is interpreted as "stretched" and must therefore be "unstretched" to obtain $P_*(1)$. This is done by so called squash-function:

$$squash(x) = stretch^{-1}(x) = \frac{1}{1+e^{-x}}$$

For each node an individual error is calculated as follows:

$$error_m = x - P_m(1)$$

The weight adjustment is now defined as:

$$w_m = w_m + error_m \times s_m \times \eta$$

With $0.002 < \eta < 0.01$ being so called "learning rate", where η can be set to a fixed value or it can be modulated by using specifically designed algorithms. Such algorithms usually gradually decrease η. Obviously the higher η is the stronger the weight adjustment. The reason for that parameter is the fact that if η would equal 1 then the weights would be "perfectly adjusted" for the current epoch in such a way that the same input again would lead to a "perfect prediction" of 0 or 1. This is not what is intended because all previous weight adjustments would be "overwritten" and there would be no memory of previously accumulated knowledge. The reason for usage of s_m in weight adjustment would reach beyond the purpose of this paper and will not be discussed.

At last, a flexible and effective way is found to combine multiple predictions by an intelligent moderation of weights. Such AI methodology can be pushed even further by using more complex NN's or NN models, but the basic principle will likely remain the same.

3 Optimizations

PAQ can be made format specific by activating a specially designed set of models for specific file formats. For example there is a special model for JPEG files which takes advantage of knowing that JPEG compression was applied. It reverses the calculation of so called "discrete cosine transform coefficients" used in JPEG Format and compresses them again using its own methods. PAQ also supports preprocessing data. This is particularly useful in text files where words can be substituted with an index based on a standard dictionary.

4 Evaluation

A comparison with well known data archivers leads to no surprises except with random files:

	TEXT	BMP	ZIP	JPEG	MP3	RANDOM
PAQ8O	19.95%	30.86%	96.31%	83.15%	94.13%	100.060%
7ZIP	28.04%	68.29%	98.38%	100.1%	98.23%	100.007%
ZIP	34.6%	76.68%	98.61%	99.89%	98.61%	100.001%
RAR	30.66%	37.93%	98.69%	100.2%	98.18%	100.025%

Figure 5: Comparison between PAQ version 8O and commonly used archivers

To ensure randomness, the random file was a 10 MB sequence of numbers downloaded from the Quantum Random Numbers Server at The Australian National University. The mp3-file was the German national anthem, the jpeg file was a 512x512 px cut of the Lena-image, which is commonly used as a test file in image processing. The bmp-file was the converted from the jpeg file, the zip file was an archived version of the operating system ReactOS 0.3.14 and the text file was a collection of all Shakespeare works downloaded from the Gutenberg Project.

By information theory, it is extremely unlikely that a random file can be compressed. It should become lager in all cases. No surprise so far, but PAQ8O turned out to be the worst in compressing random data, the reason being that PAQ compresses the entire data stream without checking if prediction is helpful to compression at all. Since the data seemed truly unpredictable, most predictions were very inaccurate independent of how they were weighted. This results in an arithmetic code that is larger than the original data most of the time. Other archive formats tend to recognize such data and can use that information to achieve better results, for example by skipping compression. That directly suggest another potential for improvement in PAQ.
PAQ explores new compression limits while running on hardware that sometimes needs hours or even days to finish the task while often using more than 1GB of memory. This, and the fact that decompression uses the same amount of time and resources, makes the algorithm unsuitable for everyday use. The main advantage is, that PAQ reaches best compression compared to other algorithms, but at cost.

The flexible plug in-like model architecture allows for the compression being very general, but also very specific at the same time. It can be standardized easily and being open source enables potential for wide spread application. In addition, developers believe that PAQ is not encumbered by any patents and recent developments like the ZPAQ standard made some newer versions backwards compatible which was not the case before.

Despite being mostly experimental and suboptimal for every day use, PAQ might have potential for the space industry, where data transmission is expensive and takes place with very limited bandwidths. For example NASA's Opportunity mars rover can send data at rates of 20 MB per hour. To the best knowledge of the author, there is no research to improve the ICER image compression algorithm which is the current standard for all NASA rovers. The author suggests, that PAQ might inspire new ideas here.

5 Summary and Outlook

When understanding the algorithm, one idea arises to improve it once more: All the models and contexts are done by limited human thinking processes. There is no way to generate models automatically and deterministically. The author's idea is that such models could be evolved by using a deterministic evolutionary algorithm. The resulting model would be very specific for a particular file and could evolve during the compression process to possibly predict better than all other "hard coded" models. On the other hand, implementing such optimization might significantly slow down the compression process.

PAQ is an advanced algorithm that improves PPM by having many different models, and many flexible contexts. This allows to use much more world knowledge and make a more accurate prediction most of the time. Like PPM, PAQ also uses arithmetic coding as its standard but because of its usage of multiple models, it has to find a way to combine them into one single prediction again. Calculating such final prediction is called "context mixing". The single outcome then can be used as usual to perform arithmetic coding.

At birth of PAQ, context mixing was performed with fixed model weights but evolved through adaptive weighting to advanced mixing methods inspired by AI algorithms. It was shown that adaptive model weighting is consistent with the intention of favoring successful models.

Although there is hardly a compression algorithm that is more effective, there

are still improvement ideas such as skipping "incompressible" parts and evolving new file specific models instead of inventing new ones directly by human hand.

PAQ is very slow and consumes a lot of hardware resources. Despite it's experimental nature, if data transfer is very slow and very expensive, usage of PAQ might be well justified. Considering Kolmogorov's findings it will be hard if not impossible to ever tell when compression limits are finally reached.

References

[Bai73] Bain, Alexander: *Mind and Body. The Theories of Their Relation.* London : International Scientific Series, 1873

[Haw04] Hawkins, Jeff: *On Intelligence.* Times Books, 2004. – ISBN 978–0805074567

[Hut05] Hutter, Marcus: *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Berlin : Springer, 2005. – 300 pages S. http://dx.doi.org/10.1007/b138233. http://dx.doi.org/10.1007/b138233. – ISBN 3–540–22139–5

[Kol65] Kolmogorov, Andrey: *Problems of Information Transmission.* Bd. 1. Taylor & Francis, 1965. – 1–7 S.

[Lak12] Lakemeyer, Gerhard: *Introduction to Artificial Intelligence.* 2011/2012. – Lecture Material

[Nim12] Nimpa, Junior L.: Huffman-Codierung, Arithmetische Codierung. In: *Seminar on compression algorithms* (2012)

[PAQ] *PAQ compression website.* http://mattmahoney.net/dc/

[RHW85] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning Internal Representations by Error Propagation / University of California. 1985. – Forschungsbericht

[Sew12] Sewing, Nils: PPM - Prediction by Partial Matching. In: *Seminar on compression algorithms* (2012)