Evaluating the CU-tree algorithm in an HEVC encoder

VLADIMIR GROZMAN
Evaluating the CU-tree algorithm in an HEVC encoder

Degree Project in Computer Science and Communication
M.Sc. program in Computer Science
KTH, Stockholm, Sweden

Vladimir Grozman

Published: Oct 29, 2015

Project provider: Ericsson
Ericsson supervisor: Per Wennersten
KTH supervisor: Mårten Björkman
Examiner: Olof Bälter
Abstract

CU-tree (Coding Unit tree) is an algorithm for adaptive QP (quantization parameter). It runs in the lookahead and decreases the QP of blocks that are heavily referenced by future blocks, taking into account the quality of the prediction and the complexity of the future blocks, approximated by the inter and intra residual. In this study, CU-tree is implemented in c65, an experimental HEVC encoder used internally by Ericsson. The effects of CU-tree are evaluated on the video clips in the HEVC Common test conditions and the performance is compared across c65, x265 and x264. The results are similar across all encoders, with average PSNR (peak signal-to-noise ratio) improvements of 3-10% depending on the fixed QP offsets that are replaced. The runtime is not impaired and improvements to visual quality are expected to be even greater. The algorithm works better at slow speed modes, low bitrates and with source material that is well suited for inter prediction.

Sammanfattning

En utvärdering av algoritmen CU-tree i en HEVC-kodare

CU-tree är en algoritm för adaptiv QP. Den körs under framåtblicken (lookahead) och minskar QP för block som refereras av många framtida block, med hänsyn tagen till prediktionens kvalitet och de framtida blockens komplexitet, approximerat av inter- och intra-skillnaden. I denna studie implementeras CU-tree i c65, en experimentell videokodare som används internt på Ericsson. Effekterna av algoritmen utvärderas på videoklippen i HEVC Common test conditions och prestandan jämförs mellan c65, x265 och x264. Resultaten är liknande i alla videokodare, med genomsnittliga PSNR-förbättringar på 3-10% beroende på vilka fasta QP-offsets som algoritmen ersätter. Körtiden påverkas inte nämnvärt och den subjektiva kvaliteten förbättras troligen ännu mer. Algoritmen fungerar bättre med långsamma hastighetsinställningar, låg bitrate samt videoinnehåll som lämpar sig väl för inter-prediktion.
Acknowledgements

To Per Wennersten, my supervisor at Ericsson, who did all the work while I just sat there and watched.

To Mårten Björkman, my supervisor at KTH, who assured me that everything was going to be OK.

To Thomas Rusert, my boss at Ericsson, who inquired as to how it was coming along.

To Ericsson itself, for actually paying their thesis workers.

To my friends and family, who still have no idea what I have been doing for seven months:
— So, what is your thesis about?
— Well... it has to do with video codecs.
— Sorry, with what?
— Never mind.
# Table of Contents

1 Introduction .................................................................................................................................................. 1  
2 Background ................................................................................................................................................ 2  
  2.1 History of video coding standards ........................................................................................................... 2  
  2.2 The structure of a video clip ....................................................................................................................... 3  
  2.3 Video compression fundamentals ............................................................................................................. 3  
  2.4 What is the role of the encoder? ................................................................................................................ 6  
  2.5 Quality metrics .......................................................................................................................................... 6  
  2.6 Rate-distortion optimization .................................................................................................................... 7  
3 The ideas of CU-tree ..................................................................................................................................... 9  
  3.1 Related work ........................................................................................................................................... 9  
  3.2 The MB-tree algorithm ............................................................................................................................ 10  
  3.3 The block structure in AVC and HEVC .................................................................................................... 11  
4 Method .......................................................................................................................................................... 13  
  4.1 Objective ................................................................................................................................................ 13  
  4.2 c65 vs x265 ............................................................................................................................................ 13  
  4.3 Implementation details ............................................................................................................................. 15  
  4.4 How to compare encoder performance ................................................................................................... 16  
  4.5 The experiment ....................................................................................................................................... 16  
5 Results ......................................................................................................................................................... 18  
6 Discussion .................................................................................................................................................... 19  
  6.1 Behavior ................................................................................................................................................ 19  
  6.2 Perceptual considerations ........................................................................................................................ 20  
  6.3 Possible improvements .............................................................................................................................. 21  
  6.4 Ethical, social and sustainability aspects ................................................................................................ 21  
7 Conclusion .................................................................................................................................................... 22  
8 References .................................................................................................................................................... 23  
Appendix I: Acronyms and abbreviations ..................................................................................................... 24  
Appendix II: Detailed results ........................................................................................................................ 25
1 Introduction
There is no need to introduce digital video – we use it every single day. However, uncompressed digital video, even at low resolutions, is unfit for distribution because of the huge files sizes involved. Fortunately, moving pictures contain a large amount of redundancy, which make them easy to compress. Over the past decades, several tools and standards have been developed for this purpose, one of the latest being the High Efficiency Video Coding (HEVC) standard.

In the hunt for ever-greater compression efficiency, many tricks have been employed to squeeze out an extra few percent – indeed, most of the gains in compression efficiency achieved in the last 20 years are down to a combination of many small features, rather than a few large ones.

In this paper, I shall explore one such feature, the Coding Unit tree (CU-tree), which is an algorithm for optimizing the quality in different areas of a picture based on temporal dependencies. This algorithm was introduced by Jason Garrett-Glaser in 2009, although similar ideas have been around since earlier. In his paper, Garrett-Glaser describes the algorithm as implemented in the x264 video encoder and presents experimental results on selected video clips [1]. For the purpose of this paper, I have implemented the algorithm in a different video encoder using the newer HEVC video coding standard and evaluated it on a standardized set of video clips in order to answer three basic questions:

1) Does the algorithm still work in a different encoder that is based on a different coding standard?
2) What results does the algorithm achieve in a standardized testing environment?
3) How does the algorithm perform under varying circumstances, such as different source material and quality settings?

I have also performed several other tests to examine the algorithm’s behavior and see how it could be improved. My testing shows that CU-tree is mostly successful in improving compression efficiency, although its performance depends heavily on the source material as well as the chosen quality and speed settings. It is likely worth implementing in an HEVC encoder.

This report is structured as follows: Chapter 2 contains a quick introduction to the field of video coding. Chapter 3 delves into the context and background of the CU-tree algorithm. Chapter 4 presents the experimental setup, methodology and relevant practical details. The results, discussion and conclusion follow in the next three chapters.
2 Background

Video coding is a subtopic in signal processing, which is somewhat of a niche in computer science. Because this paper is aimed at computer scientists in general, we first need to explain the basic concepts and other required knowledge. For this reason, the background is split in two different chapters. This chapter contains a condensed lecture on video compression, while the next one deals with the more specific context of this study.

In this chapter, we shall start by taking a brief look at the history of video coding standards. To explain what has changed between the different milestones, this section will use several concepts that will be introduced in the next two sections, where the fundamental parts of the video encoding process are explained. We will then take a look at the role of the encoder application in achieving the best possible video quality, at which point we will need to define what we mean by quality. The information in this Background is based on the book *High Efficiency Video Coding* by Matthias Wien [2] unless indicated otherwise.

2.1 History of video coding standards

The double name of the current video coding standard, HEVC/H.265, reflects the historical development of video coding since the early 1990s. The International Telecommunications Union (ITU) has been publishing standards for real-time video communication since the 80s, while the International Standardization Organization/International Electrotechnical Commission (ISO/IEC) has been making video coding standards for all other purposes, such as broadcasting and distribution of pre-recorded video content.

ITU published the very first video coding standard in 1984. It was called ITU-T H.120 “Codecs for videoconferencing using primal digital group transmission” but was never widely adopted. Instead, the first video standard with a major impact was ITU-T H.261 “Video codec for audiovisual services at $p \times 64$ kbit/s”, released in 1988. This was the first hybrid video coding standard and contained all of the basic elements used in today’s standards. For reasons of backward compatibility and largely expired patents, it is still in use today.

Meanwhile, the Moving Picture Experts Group (MPEG) was developing a more general-purpose standard for the ISO/IEC, which resulted in MPEG-1 being released in 1993. The video part of this standard built on H.261 but added some very important tools such as B-pictures and half-pel motion prediction. The MPEG-1 standard also contained several other parts, chief among them being the audio layer 3, informally known as the ubiquitous MP3. MPEG-1 was later developed into MPEG-2, which was published in 1995 and also adopted by the ITU as H.262, creating the first joint video coding standard, and was widely adopted in DVDs and digital video transmission. Following this, the paths of the two organizations temporarily diverged once again, with ITU-T H.263 being published in 1996 and MPEG-4, the video part of which largely built on H.263, saw a first release in 1999. These standards introduced many new features such as error correction and better VLC design. An extension to MPEG-4 allowed for quarter-pel motion prediction accuracy.

Realizing that their standards had largely converged despite their differing goals, ITU and ISO/IEC have developed new standards jointly since the 2000s. The Advanced Video Coding (AVC) standard was released in 2003, with ITU calling it the ITU-T H.264, while ISO/IEC called it MPEG-4 part 10. Besides the increase in compression efficiency, it featured a simplified design and improved network friendliness due to a separate network abstraction layer. High Efficiency Video Coding (HEVC), which is used in this paper, is similarly known simultaneously as H.265 and MPEG-H part 2 and was released in 2013. It has better support for higher video resolutions and encoding parallelism and requires only
about half the bitrate of a similar-quality AVC video. There are also other video coding standards
developed by other organizations, such as Google’s VP9, which is largely used on YouTube. However,
few have become as ubiquitous as those made by the ITU and the ISO/IEC.

2.2 The structure of a video clip
Before even getting started on how video can be compressed, we have to define what it is. A video clip is fundamentally a sequence of pictures or frames, which are equivalent for any purpose but interlaced video, which we will not consider here. Each picture is a matrix or a set of matrices of samples or pixels (sometimes abbreviated to pels), although it should be noted that these do not necessarily correspond to pixels on a screen: a video may be stretched or squeezed when displayed. Additionally, the samples may have a non-square shape even at the original resolution.

A sample contains intensity values for the various color channels. For monochrome (e.g. black and white) video, only one value is needed, while ordinary color video has three channels. These are not the regular RGB channels; instead, a YCbCr color space is used. Y is the luma (luminance) channel and represents the overall light intensity, the same value as that in a monochrome video. The Cb and Cr are chroma channels, which represent two different colors as the difference between the luminance and that color. With these three components normalized, the RGB values can be obtained using a simple change of basis.

There are historical reasons for employing the YCbCr color space in video coding that have to do with backward compatibility, but the main reason is that the human visual system (HVS) is more sensitive to structure and pattern than it is to color. It thus makes sense to store the former with a higher fidelity than the latter. This is achieved using chroma subsampling. The most common scheme is to subsample the chroma channels by a factor of two in each dimension, resulting in two chroma samples (one Cb and one Cr) for every four luma samples, effectively halving both horizontal and vertical color resolution.

In order to play a video, the constituent pictures must be displayed in quick succession, normally at least 24 pictures per second. Every video clip contains information about the frame rate at which it is to be played, which is normally the same rate at which it was recorded or animated. This is usually written in Hz or fps (frames per second), and most videos play at between 24 and 120 fps.

2.3 Video compression fundamentals
Video compression works by eliminating redundancy and irrelevancy. Redundant information is already present somewhere else in the file and can be obtained from there instead of storing it again. Since the changes between two consecutive pictures are typically very small, most of the information in the second picture is redundant. Removing redundancy results in lossless compression.

Unlike redundancy, irrelevancy is a relative concept. When removing irrelevant information, we start by removing details that are unnoticeable to the HVS. However, in order to achieve a target bitrate/file size, we may have to start removing information of greater importance, which noticeably degrades the video quality. In any case, we try to remove information in such a way that the quality of the resulting video is maximized.

Removing redundancy is obviously better than removing irrelevancy, since the video quality is unaffected. However, there is only a limited amount of redundancy and while it is rather easy to find some, it becomes increasingly difficult the more we try to squeeze out. Thus, every time a video is encoded, there is a tradeoff between three parameters: quality, bitrate and encoding runtime.
Not all of the redundancy that is found between pictures can be eliminated. If every picture would refer to the previous one in a long chain all the way back to the very first picture, it would be impossible to start playing a video in the middle without first decoding all the pictures that were skipped. To solve this problem, we need to introduce the concept of picture types. There are three basic types of pictures: an I-picture does not reference anything outside of itself – it contains all of the data needed to decode it. P-pictures only reference previous pictures, while B-pictures can reference previous as well as subsequent pictures. These pictures are usually arranged in a fixed, repeating order, e.g. I-B-B-P-B-B-P-B-..., where the P-pictures reference only previous I/P-pictures, never referencing past an I-picture. Similarly, B-pictures may only contain references to pictures between the enclosing pair of I/P-pictures. Thus, if we want to start playing from a specific offset, we only need to:

1. Find the nearest preceding I-picture and decode it.
2. Follow the chain of P-pictures to decode the enclosing pair.
3. Decode all the necessary pictures between the enclosing pair to get to the picture that we seek.

In practice, many video players choose to perform only step 1 and start the playback from an I-picture. The closed interval between two subsequent I/P-pictures is called a group of pictures (GOP). Confusingly, an interval between two I-pictures – which is normally much longer – is also known as a GOP. Unless stated otherwise, the former definition is used henceforth.

When data is removed from one picture and instead inferred from another, it is called inter-picture prediction. The way this happens in practice is that each picture is divided into fixed or variable-sized rectangular blocks. For each block, the encoder looks through a set of other pictures to find an area of the same size that most resembles that block, a process termed motion estimation. This area does not have to be block-aligned; indeed, it does not even have to be sample-aligned. The first standards only allowed for full-pel precision, where all referenced areas were sample-aligned, while later ones allowed for half-pel (MPEG-1) or even quarter-pel precision (MPEG-4), creating the referenced area using interpolation. This gave a big boost to encoding efficiency, because an object typically does not move an integer number of samples between pictures. A block can also reference two areas in different pictures, e.g. one from the previous and one from the next picture, the resulting prediction being a weighted average of these. This is called bi-directional prediction or simply bi-prediction. Only two-dimensional translation is considered. Although real movements can contain e.g. rotation, zooming or camera movement, these higher-order models would introduce too much complexity to be justified. The relative displacement of the referenced area from the original block is called a motion vector. A motion vector always points towards the reference, which means that it points in the direction of the actual motion when a future picture is referenced and in the opposite direction when the referenced picture is in the past.

There is also intra-picture prediction, which is used exclusively in I-pictures (hence the name) but can also be used in individual blocks of other pictures when this gives a better result. Here, data is inferred from other areas in the same picture. Specifically, it is extrapolated in one of many available modes from the row and column of pixels immediately above and to the left of the block, since picture data is decoded in sequence and data from other directions is not available at the time when the block is being decoded.

Normally, neither inter nor intra prediction can predict a given block perfectly. Depending on the targeted bitrate, the difference might be small enough that it can be skipped. In all other cases, we have a residual or prediction error that is a candidate for irrelevancy reduction. Now, simply
removing individual sample values from the residual would be far from optimal and would result in visible discrepancies. Instead, we have to find which information is the least relevant at a higher level, and then either remove it or decrease its accuracy.

First, the residual is transformed by effectively multiplying it with a transform matrix and its transpose. The specific matrices used in HEVC are Discrete Cosine Transform (DCT) II and Discrete Sine Transform (DST) IV; other standards use similar matrices. This has the effect of converting the sample values into another matrix of the same size, containing amplitude values of two-dimensional sine or cosine waves of decreasing wavelength. A block is thus converted into a weighted average of “base blocks”. For DCT, the base block that corresponds to the top left sample value is flat white. One step down is a vertical gradient from white to black, while a step to the right is a similar horizontal gradient. One diagonal step from the corner, there is a base block with two opposite white corners and two black ones, with a two-dimensional gradient in between. Further base blocks follow the same pattern but with increasing frequency, see Fig 1. The resulting weights are called transform coefficients.

Now, since large parts of most pictures do not contain a lot of fine details, the weights of some of the high-frequency base blocks will be close to zero and can be removed, resulting in fewer values to store. The values that we still choose to store are in turn quantized, i.e. have their accuracy or resolution reduced. While the original sample values may have had e.g. 256 different values for 8-bit per channel video, the transform coefficients are put on a scale that has only perhaps 64 or 16 steps in order to further reduce the bitrate. How much accuracy is kept is decided by the quantization parameter (QP), which can be specified at the picture or at the block level, the latter being referred to as adaptive QP. A higher QP value indicates stronger quantization, resulting in a lower bitrate.

When a block is predicted, the residual still contains the same number of values – unless it is skipped. After transformation, some of these may be removed, but this still does not save a lot of space. Quantization is one of the most important steps when it comes to actually lowering the bitrate, but it still does not fully utilize the biggest gain achieved during prediction and transformation, namely that most values will be very small or otherwise predictable. This is where entropy coding comes in. Using a fixed or adaptive probability model for the quantized values – with a strong concentration around the small values – we create a system of variable length codes (VLCs) in which the most
probable values take up less space than the ones that are less likely to occur. *Arithmetic coding* is a further development of VLCs. The probability models are specified in the various standards. Entropy coding is used not only to compress the quantized values, but can also be used on all other data, such as the motion vectors. Since motion vectors of neighboring blocks are likely to point in roughly the same direction, there is space to save here as well.

The coding scheme described above is known as *hybrid video coding*, since it combines inter-picture prediction with transform coding for the residual. This high-level methodology has been used for the last 25 years, but the individual building blocks have been successively refined, which has had an enormous effect on compression efficiency.

2.4 What is the role of the encoder?

A video coding standard specifies the overall structure of an encoded video file, as well as the syntax of the structural elements. Deciding the actual contents of the file is up to the encoder – an application that implements the standard and performs the compression. It needs to make an enormous amount of decisions during the encoding process to try to achieve e.g. the best possible quality for a given bitrate, which means that different encoders with different parameters can achieve wildly different results from the same input file. These decisions include:

- Which GOP structure to use. It is usually good to insert I-pictures on scene changes, since prediction would be ineffective anyway. The structure not only affects the possibility to start from a certain offset, but also determines the buffer requirements of the decoder as well as the transmission delay for live video.
- How to split each picture into blocks. Large block sizes allow for a more compact representation of large homogenous areas with uniform movement, while small block sizes are better for small objects or complex motion.
- How to perform motion estimation. It is unfeasible to consider all the possible vectors, so we have to limit the search window (or the maximum vector length). Even within this window, it is usually better to perform some kind of hierarchical search than to test all the candidates.
- Whether the residual is small enough to skip.
- Which QP values to use at file, picture and block levels.

Many of these decisions involve quality-bitrate tradeoffs. In order to know which decisions to make, the encoder needs an actual target function to optimize, which would tell it whether a particular quality-increasing measure would be worth the extra space needed. This process is called *rate-distortion optimization* (RDO). However, before going any further, we need to quantify the concept of quality.

2.5 Quality metrics

We want to maximize quality when doing video encoding, which is problematic, because quality is an inherently subjective concept. Indeed, a formal, statistically rigorous subjective test with a large enough number of test subjects is the only way to achieve anything resembling a “correct answer” as far as quality measurements are concerned. This is obviously not always feasible, especially inside an encoder, which is the reason why various objective measurements have been developed. In this context, quality instead refers to distortion – i.e. difference between original and coded video – or rather a lack thereof.

The simplest distortion measure is the *mean/sum of squared error* (MSE/SSE) and the related *peak signal-to-noise ratio* (PSNR). The sum and mean of the squared error work, just as the name implies, by simply adding together or averaging the sample-wise squared difference in values. PSNR is just the
MSE, inverted, normalized with regard to the maximum possible value (255 for 8-bit video) and placed on a logarithmic decibel scale. It is easy to calculate, and an increase in PSNR, all other things being equal, normally indicates an improvement in subjective quality, as seen in Fig 2.

Distortion is closely linked to the residual. During motion estimation, we are looking for a motion vector that minimizes the residual, which is the difference between the predicted and the actual block. Thus, similar distortion measures may be used. However, we are not actually looking for the residual with the smallest squared values, but the one that contains the smallest values after transformation, since they are the ones that will be stored in the resulting video file. For this reason, another measure is sometimes employed, the sum of absolute transformed differences (SATD). Instead of squaring the values, they are transformed with the Hadamard transform, which is much simpler than the DCT or the DST but still gives a good approximation of the size.

Returning to distortion measurement, the problem with PSNR is that it does not consider the HVS and can thus favor small but immediately visible errors over bigger errors that are easily overlooked. A simple example would be shifting all samples in the picture by a certain amount vs. shifting only the left half of the picture by the same amount. Several higher-order error measurements have been devised in an attempt to solve this issue; the most widely used being the structural similarity (SSIM) metric [3]. It assumes that the HVS is highly specialized towards extraction of structural information. By structural information, the metric denotes such features that are independent of local average luminance and contrast. The metric works by extracting the sample-wise average luminance (average value) and contrast (standard deviation) of the difference between two pictures. The structure is what remains when the difference is normalized with regard to these two values. Luminance, contrast and structure are then combined to give a single measure of the similarity.

2.6 Rate-distortion optimization
We now have a numerical measure for quality. In order to make decisions pertaining to the quality-bitrate tradeoff, we also need to know how much effect on the bitrate a certain decision will have. This is impossible to measure exactly, in part because of the adaptive probability model that is used in the entropy coding, the result of which is that the amount of space needed actually depends on the surrounding data. However, various approximations can be used instead. Indeed, we have the same problem with the quality – all the decisions regarding the current picture also influence the effectiveness of future prediction, but this effect is usually ignored.
We can thus approximate the quality and bitrate impact of a coding decision. Having these, we can go ahead and create a cost function:

\[ C(b, d | \lambda) = D(b, d) + \lambda \times R(b, d) \]  

where \( b \) is the block under consideration, \( d \) is the decision to be made, \( C \) is the cost to minimize, \( D \) is the distortion in \( b \) introduced or eliminated by \( d \), \( R \) is the corresponding change in bitrate, while \( \lambda \) is an application-dependent Lagrangian weighting factor that also serves to specify the desired tradeoff. Since the tradeoff – how much space we are prepared to sacrifice for a given quality improvement – depends on the desired quality, \( \lambda \) is usually calculated as a function of the QP value.

RDO is often the main performance bottleneck of the video encoding process. By deciding how much time to spend here, i.e. how many possible decisions to evaluate, we can trade encoding time for a shift in the quality-bitrate tradeoff curve. Many encoders feature selectable *speed modes or presets* to reflect this secondary tradeoff. These can also change other parameters, such as how many different motion vectors to test.
3 The ideas of CU-tree

In this chapter, we start by examining three examples of previous work that are based on the same concept as CU-tree, which is the algorithm that we are studying. MB/CU-tree itself is introduced next, followed by some important differences between the AVC and HEVC standards and how this might affect the algorithm’s performance.

3.1 Related work

The main idea behind the MB-tree/CU-tree algorithm is that instead of ignoring various decisions’ effect on future picture quality, we can take these temporal dependencies into account. Before diving into the algorithm, it is beneficial to look at other variants of this idea. They are all implemented in AVC encoders, similar to MB-tree.

In [4], Schumitsch et al. propose the following idea: assume at first that all the motion vectors and QP values are given. When making any decisions pertaining to a certain block, we thus have a full list of all other blocks affected by this decision. Aside from the decisions listed in section 2.4, there is one decision that is normally rather implicit – that of the actual transform coefficients. Intuitively, we should simply take the values obtained from the transform operation, since these are the values contained in the original picture. Nevertheless, could we perhaps change these values, introducing some small errors but in turn helping future prediction? The authors propose taking a number of consecutive pictures (e.g. a GOP) and jointly optimizing all of the transform coefficients in these. By putting all of the transform coefficients into a giant array, the dependencies can be modeled as a single matrix. Using some approximations, this can be used to extend (1) to a quadratic program, which can be solved “efficiently” (in polynomial time).

One notable problem with this approach is one that also affects other algorithms that try to use inter-picture dependencies in a similar way. When a decoder decodes an inter-coded block, it takes the area in the reference picture indicated by the motion vector, a process known as motion compensation, and adds the residual. However, the decoder does not have access to the original video with the original reference picture; it can only perform motion compensation based on the approximated reference picture that is has itself previously decoded. Errors obtained this way would accumulate unless the encoder compensated for them. For this reason, the encoder actually performs motion estimation based on decoded pictures. Before using a picture as a reference, it must be fully encoded, after which it is decoded once again and placed in the decoded pictures buffer, which the motion estimation uses. The problem with using motion vectors of future pictures to make decisions about the current picture is that these decisions are likely to change the optimal vectors, i.e. a chicken-and-egg problem.

If we, as previously, consider the motion vectors to be fixed, they will no longer be optimal after changing the transform coefficients. To solve this, the authors propose an iterative method: optimize the coefficients based on initial prediction, predict new vectors based on these coefficients and optimize the coefficients again using the new vectors. They achieved PSNR improvements of 0.7 - 1.0 dB in two different test videos. Because an external quadratic program solver was used, the authors admit that their algorithm is “not ideal for real-time encoding or low-delay encoding”.

Mohammed et al propose another approach in [5]. They propose allocating priorities to each block based on how many future blocks it is referenced by. In order to do this, they keep a reference counter on each sample. A two-pass encoding scheme is used, where the video is first encoded once and the motion vectors are analyzed to determine block priorities. The whole video is then re-encoded using these new priorities.
Priorities are determined as follows: first, the motion vectors are used to determine how many future samples are referenced by each sample. The counters are kept at the sample level because vectors are not block-aligned, but all the counters in each block are averaged in the end to produce a single reference counter for the block. The resulting range of reference counter values is divided into nine subranges of equal size and the blocks within each subrange are given QP offsets of -4 to +4. This gave an average improvement of 0.4 dB across five test videos, with a much more reasonable performance penalty than the previous approach.

Amati et al. build on this idea in [6], but additionally consider how well a block predicts the blocks that refer to it. They prove theoretically that assigning a lower QP value to blocks that predict other blocks with a lower residual improves PSNR. They then use various heuristics derived from their theoretical results to optimize both motion vectors and QP values in a two-pass encoding scheme. However, they do not consider B-pictures or subpixel motion estimation, which severely limits the practical use of their algorithm. Compared to encoding with these same features disabled but without the algorithm, the authors achieved a 1 dB improvement in one test video and no improvements in two other videos.

3.2 The MB-tree algorithm

The algorithm that is used in this paper was devised by Garrett-Glaser and is similar to the ideas of Amati and Mohammed, but differs in a few important ways [1]. It has a slightly different way of measuring how “important” a block is and it supports subpixel motion estimation and B-pictures. Also unlike the previous two algorithms, all measurements are kept at the block level.

Instead of two full encoding passes, MB-tree uses a simpler lookahead pass. This pass runs in parallel to the main pass but a few pictures or GOPs ahead of it. It avoids many of the difficult decisions such as block partitioning and instead assumes a fixed 16x16 block size. It also operates on a subsampled picture with its width and height cut in half, further improving performance and resulting in 8x8 blocks. For the purpose of this algorithm, it only performs two tasks:

1. Motion estimation for the fixed-size 8x8 blocks and inter residual calculation.
2. Intra prediction and intra residual calculation.

Using the motion vectors, the algorithm creates a weighted directed acyclic graph of dependencies, hence the “tree”. Since the vectors are not block-aligned, each vector is split in up to four edges whose weights are specified by the portion of the referenced area that is contained within the block it points to, thus always summing up to one. If a block references two areas in two different pictures, it can have up to eight outgoing edges, but their weights still sum up to one in accordance with the weighting of the prediction vectors.

Now, the idea is as follows: obviously, a block is more important if it is referenced by many other blocks, i.e. has many ancestors in the graph. Moreover, it is more important if it predicts these blocks with a lower residual. However, unlike the previous two algorithms, we now also consider how difficult those blocks are to predict independent of the current block, and this is where the intra prediction comes in. If a block is not only well predicted by the current block but also by intra prediction, we have not really gained that much by using inter prediction and the current block should not benefit as much from this dependency.
On the graph, the algorithm works as follows:

1. In order to determine how much a certain block should contribute to the importance of its reference blocks, define the function

   \[ propagate\_fraction = \max(0, 1 - \frac{\text{inter\_residual}}{\text{intra\_residual}}) \]  

   This means that the contribution should increase with increased intra and decreased inter residual. Specifically, if the inter residual is 30% of the intra residual, we assume that 70% of the information in the current block is sourced from the reference blocks. Obviously, if the inter residual is larger than the intra, then this particular inter prediction is useless.

2. Start from the vertices with no outgoing edges, which correspond to the blocks of the “top-level” B-pictures that are never referenced by other pictures (see section 4.2). These vertices are assigned their intra residuals as weight.

3. Whenever all the parents of a vertex have been weighted, multiply the weights of each parent with their respective \(\text{propagate\_fraction}\) and the weight of the connecting edge. To get the weight of the vertex, sum up all the resulting values and add the intra residual.

In practice, there is no need to build an explicit graph. Instead, the algorithm starts with the unreferenced B-pictures in the last GOP of the lookahead and works backwards one picture at a time, always picking one that has got all its reference pictures processed already. Finally, when we want the QP offset for a particular block, it can be calculated as

\[
QP\_offset = strength \times \log_2 \left( \frac{\text{weight}}{\text{intra\_residual}} \right)
\]

where \(strength\) is an experimentally derived parameter of the algorithm. There is no need to calculate (3) for all the pictures that the algorithm processes. Normally, the lookahead is a fixed number of pictures or GOPs ahead of the main pass, but the whole algorithm is rerun e.g. when the main pass has finished coding a GOP, so we only need to obtain the offsets for the next GOP. All the other pictures are only processed to generate the correct weights for this GOP. When the main pass has finished processing it, the lookahead pass will have advanced and all the weights of subsequent pictures will need to be updated.

The algorithm presented above works roughly the same in AVC and HEVC. However, there are important differences between these standards that affect the algorithm’s performance – and even its name.

### 3.3 The block structure in AVC and HEVC

The block structure in AVC works by dividing every picture into 16x16 macroblocks, which can in turn be divided into 8x8, 16x8, or 8x16 sub-macroblocks, the latter two only being available for inter-coded blocks. These can in turn be similarly divided down to a size of 4x4. This is where the MB in MB-tree comes from – it is in fact a “tree” of macroblocks.

The block structure of HEVC is much more complex. Each picture is divided into coding tree units (CTUs) with a per-video defined size of 16x16, 32x32 or 64x64. These are divided into coding units (CUs) using a quadtree structure. A quadtree is a square with a power-of-two size. The square can either be left as-is or split into four squares of equal size. Each of the small squares can in turn be split further in the same way independently from the others. CU sizes down to 8x8 are allowed.
CUs are split independently into *prediction units* (PUs), which contain the prediction information and *transform units* (TUs), which contain the transform coefficients. Intra-predicted PUs are the same size as the CUs, except for 8x8 CUs, which can be split into 4x4 PUs. When inter prediction is used however, CUs can be split into PUs in eight different ways, four of which are asymmetric – see Fig 3. Splitting CUs into TUs is performed using another quadtree, also down to a minimum size of 4x4. To simplify things a bit, 4x4 inter-predicted PUs are not allowed and 4x8 and 8x4 PUs can only use uni-directional prediction.

CTUs are the only fixed-size units in HEVC, and even their size can vary between different video clips. CUs are the largest units that contain both prediction and residual data, both of which are used in the algorithm, which presumably is why it is called CU-tree when used in an HEVC encoder. However, the size of the CUs, as well as that of the PUs and TUs, can vary wildly in the main pass, which puts the lookahead assumption of fixed 16x16 blocks on its end. It is not easy to fix either by letting the lookahead perform block-size decisions, since this is one of the most expensive RDO operations. Besides, just like the transform coefficients in section 3.1, the resulting QP values would change the optimal block structure. Now, CU-tree has in fact been implemented in at least one HEVC encoder, x265. This is the successor to the x264 AVC encoder, which saw the original MB-tree implementation. It has been left mostly unchanged between these two encoders, and the difference in performance due to the change of standard has, to my knowledge, not been examined.
4 Method

4.1 Objective

This study aims to expand on the original MB-tree study [1] in three significant ways. First of all, MB-tree was implemented and evaluated in x264, an AVC encoder. This is no longer the latest video coding standard, and it is useful to know how it performs under the new HEVC standard. Although CU-tree has later been implemented in x265, its performance has not been similarly measured. Additionally, it is valuable to know how well the algorithm performs in an encoder architecture that differs significantly from x264/x265 – the target of this study is c65, an experimental HEVC encoder created and used inside an Ericsson research department. As described in section 2.4, various encoders can behave quite differently and some of the differences between c65 and x265 are described in the section below.

Second, although the original study evaluated the algorithm, this evaluation was performed on a custom selection of video clips that might not be representative. The results were mainly presented as a series of bitrate-PSNR graphs and contained few hard numbers. The current study was performed on the 19 video clips listed in the HEVC Common test conditions [7] (Random access, 8 bits), with quantified results. The performance measurements and comparisons are explained in more detail in sections 4.4 and 4.5.

Third, the original study was rather brief also when it comes to examining the non-quantifiable aspects of the algorithm, such as how it performs at different speed modes and with different types of source material. In the discussion, we will make some important conclusions about such characteristics of the algorithm.

4.2 c65 vs x265

x264 and its successor x265 are widely used, open source general purpose encoders with an extensive set of features that can be enabled or disabled to accommodate many different workloads and usage scenarios [8]. In contrast to this, c65 is a purely experimental encoder created for internal use only. At the time when this study was conducted, c65 was mostly geared towards slower speed modes, where the encoder spends a lot of time in RDO, testing many different possibilities. For this reason, only the results of the slowest mode are reported for c65.

Unfortunately, it is not possible to directly compare the various speed modes of encoders with vastly different feature sets. There is no standardized definition of what a speed mode does or how many of them there should be. When simply comparing various encoders to see which performs best, speed modes can be matched by measuring the encoding time of the same video clip on the same machine. This is not possible when comparing the implementation of a specific feature in two different encoders, especially when they do not enjoy a similar purpose and level of optimization. For this reason, the absolute results reported in this study should not be interpreted as a comparison between the various encoders. Only the changes seen between different configurations of the same encoder are significant, and these can in turn be compared between the encoders. (For reference, the ‘slow’ preset is reported for x264 and x265 in the next chapter as well as in the appendix.) Moreover, at the time when this study was conducted, several important features of c65 were not yet compatible with adaptive QP and had to be disabled; thus, the numbers reported in this study are not indicative of the performance normally achieved by c65. As a disclaimer, the purely experimental c65 encoder is in turn not representative of any current or future product offerings by Ericsson. In no particular order, here are some specific differences between c65 and x265 that are important for this study:
For the purpose of this study, all the encoders were configured to use a fixed eight-picture GOP, i.e. seven B-pictures between two consecutive I/P-pictures, and six GOPs (47 pictures) between two consecutive I-pictures. All of the encoders also use a feature called B-pyramid, where the pictures are divided into temporal layers, see Fig 4. The first layer contains all the I and P-pictures. The second layer consists of the middle B-pictures in each GOP, which predict from the surrounding pair of I/P-pictures. In x265, the top layer contains all remaining pictures, which predict from the surrounding pair of lower-layer pictures. c65, however, has another layer in the pyramid, which consists of the second and sixth B-pictures of every GOP. The top layer once again contains the remaining pictures and predicts from the surrounding pair of lower-layer pictures, which in this case are the immediate neighbors.

During motion estimation in the lookahead, c65 uses a hierarchical approach. The lookahead already works on a picture that is subsampled by a factor of two in each dimension, but c65 performs further subsampling in the lookahead. It first performs the motion estimation on a picture that is subsampled by a factor of eight (in each dimension). The motion vectors obtained in this step are used as a starting point for motion estimation in a picture subsampled by a factor of four, which in turn is used as a starting point for motion estimation on the primary lookahead picture. The motion estimation in the main pass is based, in turn, on the last vectors obtained in the lookahead. All motion estimation in the lookahead is made on 8x8 blocks regardless of picture size, so every vector in the first two levels is used as a starting point for four vectors in the next level. Motion estimation in the lookahead is not affected by the speed mode, unlike the main pass.

Lookahead motion estimation in x265 is also hierarchical, but in a very different manner. It works by using hierarchical search patterns instead of testing all vectors up to a given length. Given an initial vector length, e.g. 3 samples, it repeatedly tries to find the best vector of this length starting from the best vector found in the previous step, using a predefined geometrical pattern. After a certain number of steps, or when all the newly found vectors are inferior to the vector in the previous step, it switches to a shorter length [9] [10]. The pattern used depends on the speed mode [11]. Unlike

---

Figure 4. GOP structure in c65 (above) and x264/x265 (below), as used in this study. The different layers are color-coded. The arrows point in the same direction as the motion vectors. The dotted lines do not apply when the last picture is an I-picture.
c65, the motion estimation in the lookahead is affected by the speed mode, since it works in the same way as in the main pass.

In c65, the lookahead only uses the luma channel. One of the changes tested during the course of this study was to change it to use all the color channels. c65 also uses SSE for all the error measurements in the lookahead, while x265 uses SATD. Although SATD is more accurate, the error measurement singularly constitutes the hot spot in an encoder, and properly implementing and optimizing SATD in c65 was outside the scope of this study.

x265 allows measuring both PSNR and SSIM, but c65 only includes facilities to measure PSNR, so only PSNR measurements are used in this study. The original study on MB-tree [1] reports significantly greater improvements to SSIM than to PSNR, and there is no reason to believe that this particular fact would be subject to change between different standards or encoders. x265 also contains another algorithm that uses adaptive quantization and only improves SSIM, but it has been disabled in this study.

Note that almost none of the details about x265’s behavior are documented. Most of the information in this section is taken directly from its source code or from analyzing the output video files. Any number of details may have changed since this study was conducted, and these changes would likely not have been documented either.

4.3 Implementation details

The lookahead in c65 always works one GOP at a time and runs a configurable number of GOPs ahead of the main pass. This number has been set to three in this study, although other values were also tested. Thus, the encoding starts with the lookahead running on three GOPs, after which the main pass starts. Every time the main pass finishes coding a GOP, the lookahead runs a full GOP, after which the main pass resumes. This can be parallelized easily, which is one the main justifications for having a lookahead pass.

Before the main pass resumes, CU-tree runs on all GOPs that have been processed by the lookahead but not by the main pass, starting with the last one. The final step, where the QP offset is calculated, only runs right before a picture is encoded by the main pass. Increasing the lookahead distance would thus increase the size of the tree and the runtime of CU-tree, but would not affect the runtime of the rest of the lookahead or the QP offset calculation. The reason for increasing the distance is to include more dependencies and thus increase accuracy, however, the strength of the dependencies decreases with the distance.

The QP offsets in CU-tree are non-additive in the sense that the QP offset for a 32x32 block is not the same as the average of its theoretical constituent 16x16 blocks. This makes sense, since the QP scale is non-linear. For this reason, the QP offset is calculated separately for all possible (aligned) 16x16, 32x32 and 64x64 blocks. Although in the end only one of these will be used, they might all be needed when deciding which block size to use. Additionally, the average QP for the whole picture is calculated even when average CU-tree mode is not used (explained in section 4.5), since this value is needed elsewhere in the encoder. A strength parameter of 2.0 was used for the comparison, same as the default in the original MB-tree implementation. Other values were tested as well.

A few bugs were particularly hard to detect, especially for someone who is not experienced in the field of video encoding. Because c65 uses SSE instead of SATD in the lookahead, the square root of the errors has to be used in CU-tree. Additionally, the error measurements have to be scaled when working with subsampled pictures.
4.4 How to compare encoder performance
In section 3.1, all the authors presented their results as decibels of PSNR improvement. If this did not give you an intuitive understanding of the magnitude of their improvements, you are not alone. There are two problems with this approach: first, an improvement of 1 dB will not necessarily look the same when the starting value is 30 dB and 40 dB. Second, actually achieving a 1 dB improvement at 40 dB is significantly more expensive. Because the high-level visual data is already mostly correct, the improvements have to come from the fine details, which are much harder to compress – and to see. However, this is only a general effect: how noticeable a small PSNR improvement is and how much it costs in terms of bitrate can actually be quite random, depending on the data in the particular video and the specific coding decisions that are changed.

For these reasons, it is not actually very helpful to ask, “How much better quality can I get at the same bitrate?” when doing encoder comparisons. Instead, we should ask, “How much lower bitrate can I get while achieving the same quality?” Bitrate is linear, easy to understand, and if Encoder 1 achieves the same quality as Encoder 2 at a 10% lower bitrate, we can simply say that Encoder 1 is 10% better. Because the achieved improvements generally vary based on the chosen quality level, the industry standard is to use a Bjøntegaard Distortion Rate metric, or BD rate [12]:

1. Measure the bitrate at four quality levels and plot it on a bitrate-PSNR graph with a logarithmic bitrate scale. For HEVC, use average QP values of 22, 27, 32 and 37 [7].
2. Fit a third-degree polynomial to the four (PSNR, logarithmic bitrate) points for each of the compared encoders.
3. Integrate between the resulting curves. Pick the integration limits to avoid extrapolation, i.e. use the maximum of the lowest logarithmic bitrate values as the starting point and vice versa.
4. Divide by the integration interval to obtain the average relative bitrate improvement.

A logarithmic bitrate scale is used to prevent the high bitrate results from having a disproportionate influence on the average result. Using the polynomials obtained with this method, we can calculate not only the average improvement, but also obtain separate measurements e.g. for low and high bitrates. In addition, we can view the results graphically by plotting the curves.

4.5 The experiment
To perform a fair comparison between different encoders’ MB/CU-tree performance, we need to establish a common baseline – what the algorithm is replacing. Using the same QP values across a whole video clip is extremely inefficient, so the encoders used in this study already implement various optimizations. The simplest optimization is to use fixed QP offsets between I, P and B-pictures. The logic behind this is very simple: I-pictures are referenced more than P-pictures, which are referenced more than B-pictures. The more a picture is referenced, the more improving the quality of that picture should improve the quality of other pictures, and thus of the whole video. x264 and x265 use offsets of approximately -3 and +2 for I and B-pictures, compared to P-pictures, and this is used as a baseline in this study. Interestingly, the original MB-tree implementation does not replace the fixed QP offset for I-pictures; instead, the two offsets are added.
These two encoders can also use a different algorithm, quantizer curve compression (qcomp). The idea is to increase the QP in picture areas with high complexity and many fine details, thus decreasing quality, and lower it in low-complexity areas. This might sound counterintuitive, but there are three reasons for doing this [13]:

- High complexity often corresponds to fast motion, so a decrease in quality will be less noticeable.
- High complexity often corresponds to fast non-translational change, so it is likely that these areas are not referenced as much by future pictures.
- High complexity areas are harder to compress, so decreasing the quality here will save more space.

By default, c65 also uses fixed QP offsets, but they are calculated in a more complicated manner and are heavily optimized based on real-world testing. It should be noted that they have been optimized solely for PSNR and are not necessarily accompanied by a similar improvement to visual quality. At the request of Ericsson, the details are not disclosed in this study, but consider this as being close to the limit of what can be achieved using fixed QP offsets.

Several different ways of combining CU-tree and fixed offsets were tested in this study. To examine the effects of CU-tree more closely, the CU-tree implementation in c65 has an additional mode: average CU-tree. It works by running the CU-tree algorithm and averaging the QP offsets within each picture. This allows us to see to which extent the gains achieved by CU-tree can be attributed to the redistribution of bits between pictures, similar to the fixed QP offsets. The rest of the improvements should then be attributed to the redistribution of bits within each picture. For testing purposes, I have also added this mode to x265.

The comparison was performed on the 19 8-bit random access video clips listed in the HEVC Common test conditions [7]. Eleven encoder configurations were tested on the full set of videos:

- c65: CU-tree, average CU-tree, fixed offsets, fixed offsets from x264/x265
- x265: CU-tree, average CU-tree, qcomp, fixed offsets
- x264: MB-tree, qcomp, fixed offsets

Up to seven pictures were truncated at the end of each video to leave an integer number of GOPs—otherwise, they would have been encoded as P-pictures or using a different GOP structure. To save space, only the luma PSNR will be reported. All videos were encoded at 25 fps regardless of their original framerate, which affects the reported bitrate proportionately.

The following command line parameters were used for x264 and x265, in addition to the parameters used to control MB/CU-tree and qcomp:

```
--tune psnr --psnr --no-psy --rc-lookahead 24 -I 48 -i 48 -b 7 --b-adapt 0 --aq-mode 0 --weightp 0
```

```
--tune psnr --psnr --no-psy-rd --no-psy-rdoq --rc-lookahead 24 -I 48 -b 7 --b-adapt 0 --aq-mode 0
```

17
5 Results

A strength value of 2.0 is generally optimal and is used across all MB/CU-tree implementations in the final setup. Fixed QP offsets of +2 for B-pictures and -3 for I-pictures are added to the QP offsets generated by CU-tree in c65. For detailed results, showing bitrate-PSNR curves and relative improvements in each mode, see Appendix II. The following table lists the average improvements (bitrate reduction) achieved using MB/CU-tree compared to the other modes of the same encoder. Negative values mean that MB/CU-tree fared worse.

Table 1. Average improvements achieved using MB/CU-tree compared to the other modes of the same encoder.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c65 avg CU-tree</td>
<td>-1.4%</td>
</tr>
<tr>
<td>c65 fixed offset</td>
<td>3.0%</td>
</tr>
<tr>
<td>c65 with x265 offset</td>
<td>9.7%</td>
</tr>
<tr>
<td>x265 avg CU-tree</td>
<td>1.1%</td>
</tr>
<tr>
<td>x265 qcomp</td>
<td>7.9%</td>
</tr>
<tr>
<td>x265 fixed offset</td>
<td>9.6%</td>
</tr>
<tr>
<td>x264 qcomp</td>
<td>7.2%</td>
</tr>
<tr>
<td>x264 fixed offset</td>
<td>8.9%</td>
</tr>
</tbody>
</table>

The most important result is that MB/CU-tree performs similarly in c65, x265 and x264 on average, as well as in the individual video clips, with the exception of a few outliers. Other important results are the rather small average difference between CU-tree and average CU-tree, as well as the large difference between x264/x265 and c65 fixed offsets. If we also look at the graphs in the appendix, we see that MB/CU-tree usually delivers greater improvement at lower bitrates.

When using CU-tree, PSNR-Cb and PSNR-Cr improved significantly more than PSNR-Y, even in c65, where the lookahead only uses luma. Changing c65 to use all color channels in the lookahead further improved PSNR-Cb and PSNR-Cr without significantly affecting PSNR-Y. Increasing the lookahead distance beyond three GOPs only yielded very small improvements.

The runtime of the CU-tree algorithm and QP offset calculation is negligible in all speed modes, since they do not perform any RDO or error measurement, which is what the encoder spends most of its time doing. The lookahead also does not perform any RDO, making its runtime negligible in slow, RDO-heavy speed modes. For fast speed modes, however, the addition of a lookahead pass caused a performance penalty of up to 30% in c65. MB/CU-tree also generally performed worse at faster speed modes, with average improvements over fixed offsets in ‘medium’ and ‘fast’ modes of 9.0% and 5.2% in x265 and 8.9% and 7.9% in x264.

Several other ways of combining CU-tree with fixed offsets were tested but failed to improve the results:

- No fixed offsets
- A few fixed B/I offsets other than -3/+2
- Adding CU-tree to the existing fixed offsets in c65
- Adding (CU-tree - avg CU-tree) to the existing fixed offsets in c65, i.e. centering the CU-tree offsets of each picture around the fixed offset
6 Discussion

Given the result, the natural question is whether CU-tree is worth using in c65 as well as other encoders.

The algorithm behaves similarly in AVC and HEVC, disproving the conjecture that the more flexible block structure of HEVC would pose a problem. The algorithm also behaves similarly in two different implementations across two completely different HEVC encoders, which means that it is likely to give similar results also in other HEVC encoders.

The runtime of the main algorithm is negligible, however, the runtime of the lookahead might not be. At the slowest speed modes, the improvements of CU-tree justify the runtime of the lookahead even if the latter is not used for any other purpose. However, this is typically not the case: in c65, the vectors from the motion estimation in the lookahead are used as starting values for the motion estimation in the main pass. Even in single-threaded use, this is similar to simply using hierarchical motion estimation in the main pass, so the time spent here is not actually wasted. Given the additional possibility of parallelization, having a lookahead is usually justified even without CU-tree. Indeed, c65 already had a simpler lookahead before the CU-tree implementation, and the lookaheads in x264 and x265 perform many other tasks besides motion estimation. However, it is not always possible to use a lookahead (and thus CU-tree), e.g. when coding video for live bidirectional communication.

6.1 Behavior

There are two elephants in the room. One is the huge difference between the results obtained using fixed QP offsets in c65 and x264/x265. In the latter two, using MB/CU-tree resulted in massive improvements across almost all video clips, so using it where applicable instead of fixed offsets or qcomp is a straightforward decision. In c65, however, the decision is not as easy. Even though CU-tree gives an average improvement for most videos, some video-bitrate combinations are actually worse off. However, it should be noted that the fixed offsets in c65 are optimized for PSNR and might improve it more than they improve visual quality, while the opposite applies to CU-tree.

The other elephant is the average CU-tree. The small difference compared to full CU-tree is surprising, since it means that most of CU-tree’s effects stem from the redistribution of quality between pictures, which is what fixed QP offsets do, as opposed to the redistribution of quality to the blocks that are highly referenced within each picture. However, the result for c65 does not actually mean that redistributing quality within a picture makes it worse, only that the quality improvements obtained in this way are outweighed by the space required to store the per-block QP values, which normally constitute around 2% of the bitrate. This is especially clear since average CU-tree was executed with adaptive QP disabled in c65, but not in x265, and the latter provided better results with full CU-tree. Since adaptive QP is normally enabled in real-world usage, using average CU-tree does not improve results in practice.

The fact that most of the improvements come from the redistribution of quality between pictures is actually good news, because it means that real-world improvements should be even greater. This study only evaluated the algorithm on standard test video clips, which last only a few seconds and contain a single scene each. Normally, video clips consist of several different scenes, which means that quality can be redistributed toward scenes where inter prediction is most effective. This effect was also observed in the original MB-tree study [1].
Although some of the tested videos fared worse than fixed offsets even at low bitrates, many others had a cutoff point, where the two lines crossed and CU-tree stopped providing an advantage. There is a reason for this effect: at low bitrates, any additional bitrate from the changed QP goes towards coding the large-scale details of the block, which are likely to be real properties of the scene and move in the direction of the estimated motion vector. At high bitrates, however, additional bitrate might go towards coding camera noise as well as other fine details that might not move uniformly with the rest of the block. The former is likely to improve the quality of referencing blocks, while the latter might even result in a loss of quality.

A related effect can be observed when running in faster speed modes. For MB/CU-tree to function correctly, two things must be true about the motion vectors in the lookahead:

1. They have to approximate the main pass vectors well in order for the tree to resemble the actual referencing pattern. Enhancing one block does not improve the quality of another block if the latter ends up not referencing it after all.
2. They also have to be ‘correct’, i.e. correspond to real movement, so that improving one block would decrease the residual of the referencing blocks. If the motion estimation does not find the optimal vector and settles for something that is incorrect or only approximately correct, enhancing a block might in fact mean that the matched blocks become less similar, increasing the residual.

Since faster speed modes typically spend less time trying to find the optimal vector, it is logical that MB/CU-tree should give less of an improvement. As is clearly seen in the graphs in the appendix, the contents of the video also play a huge role. In general, anything that improves prediction, such as static or computer-generated content, also improves the performance of the algorithm. Examples of such material are news programs, animation movies and video game footage. In contrast, noisy and grainy material, as well as videos that contain fine or non-translational movement see limited improvements. Billowing grass or hair, camera noise and film grain are examples of such content.

It is surprising that the chroma PSNR improved significantly more than the luma. One possible explanation is that CU-tree reduced the luma residuals, thus freeing more space for the chroma channels within the same bitrate envelope. However, balancing the space requirements of the luma and chroma channels is a separate, complex topic, so we should not make any drastic conclusions based on this data alone. It is enough to state that the opposite – luma improving significantly more than chroma – would have been problematic, since that would have meant that luma improved, at least in part, at the cost of chroma. Fortunately, this was not the case.

There are two outliers in the results that are worth mentioning: SlideEditing and BQTerrace. These are both rather atypical video clips. SlideEditing is a screen capture of a PowerPoint session, while BQTerrace captures a slowly panning camera with little additional movement. Although these results could indicate bugs in either implementation, the ‘extreme’ nature of these clips means that they could easily be explained by any number of small differences between the encoders. A prime candidate for such a difference is the +2 fixed QP offset for B-pictures that is used together with CU-tree in c65.

6.2 Perceptual considerations

Until now, we have almost exclusively considered the PSNR. Although an improved PSNR usually signifies an improvement in perceived quality, there are a few concerns specifically with regard to CU-tree that have not yet been mentioned in this report. First, CU-tree changes the QP values within a picture in a way that is not directly related to the content of the blocks, unlike qcomp. This could
result in visible changes in quality across a picture. AVC and HEVC include complex deblocking filters to reduce the visibility of block borders, but in order to further reduce the visibility of the QP changes, a Gaussian blur could be used on the variables involved in CU-tree. In fact, this is what x264 does with qcomp [13].

Another problem is the way CU-tree redistributes space between the pictures. Unlike fixed offsets or qcomp, CU-tree improves the quality of pictures that come shortly after I-pictures and decreases the quality of pictures right before them. Since an I-picture “breaks” all the referencing chains, the pictures that come before it are referenced less than the pictures that come after it. This might result in a visible jump in quality at the I-picture, followed by gradually decreasing quality, followed by another jump at the next I-picture, and so on. This can be alleviated by decreasing the lookahead length – if it is just one GOP, this effect does not appear. A better solution, which is what x264/x265 do when configured to optimize visual quality instead of PSNR, is to treat I-pictures as P-pictures for the purpose of MB/CU-tree, which removes this effect completely.

Thirdly, similar to the problem with I-pictures, CU-tree can produce a “pre-echo”, where it decreases the quality of pictures right before a scene change. However, this effect is masked in the HVS by the oncoming scene change [14].

6.3 Possible improvements
The most obvious improvement would be to further optimize the QP offsets for the various picture types and temporal layers, just as it is done in c65 for fixed offsets. Looking at the effects of the latter, this might improve results significantly.

Another possibility, at least for slow speed modes, is to run the lookahead and CU-tree algorithm twice, using the results of the first pass to find better vectors in the second pass. This should alleviate the fact that changing QP values also changes the optimal vectors, as described in section 3.1.

As the positive effects of CU-tree diminish with increased bitrate, it might be worth to look into various solutions for detecting this, such as analyzing the relative quality of the motion prediction. Then we could either reduce the strength or turn off CU-tree completely at high bitrates.

A fourth possibility is to store all the variables of the algorithm at the sample level, just like the examples in section 3.1. This would be rather easy to implement, would not affect runtime significantly and would remove what is essentially an averaging of all the variables at each vertex in the tree. However, looking at the small difference between average and full CU-tree, this would likely not yield a significant improvement.

6.4 Ethical, social and sustainability aspects
This section is obligatory for all thesis reports at KTH.

In general, better video compression means lower storage and bandwidth requirements, leading to a decrease in power consumption, all of which is important, considering the massive amount of storage being used for video. The current study does not have any significant social effects or ethical consequences.
7 Conclusion
In this study, we have thoroughly examined the performance and behavior of the CU-tree algorithm in order to see whether it is worth using in an HEVC encoder. We have done this by implementing it inside a new encoder architecture and testing it on a wide variety of video clips under standardized testing conditions. As for the questions posed in the introduction, these have been answered as follows:

The algorithm performs similarly in AVC and HEVC, as well as in different encoders implementing these standards.

The algorithm provides a significant average PSNR improvement to the standard test video clips without impairing the runtime. Because of the structure of these clips and the greater SSIM improvements seen in the originals study [1], the improvements are likely to be even greater when it comes to subjective quality as well as for real-world videos. The PSNR improvements depend on the level of optimization of the fixed QP offsets in each encoder, but CU-tree provides a significant improvement even compared to heavily optimized fixed offsets and even when its own fixed offsets are barely optimized. More research is needed to find the optimal combination of CU-tree and fixed offsets.

The algorithm generally performs well when the motion estimation performs well. This means that the source material is well suited for inter prediction and that the encoder runs at a slow speed mode so that it has enough time to find the optimal motion vectors. Additionally, improving quality must not result in the coding of noise, which often happens at high bitrates. More research is needed to see whether these conditions could be detected automatically.
8 References


Appendix I: Acronyms and abbreviations

**AVC** and **H.264** both refer to the previous generation standard created by ITU and ISO/IEC.

**CTU/CU/PU/TU** – Coding tree unit, coding unit, prediction unit and transform unit are the spatial building blocks in HEVC.

**DCT/DST** – Discrete Cosine/Sine Transform, the two transform types used in HEVC.

**Fps** – Frames per second, also called Hz.

**GOP** – Group of pictures, i.e. the space between two subsequent I/P-pictures.

**H.*** - See **AVC**, **HEVC** and **MPEG-1**.

**HEVC** and **H.265** both refer to the current generation video coding standard created by ITU and ISO/IEC.

**HVS** – Human visual system

**ISO/IEC** – International Standardization Organization/International Electrotechnical Commission

**ITU** – International Telecommunications Union

**MB** – Macroblocks are one of the spatial building blocks in AVC.

**MPEG** – Moving Picture Experts Group

**MPEG-1, MPEG-2, MPEG-4, H.120, H.261, H.262, H.263, VP9** – Various other video standards

**MSE/SSE** – Mean/sum of squared error

**Pel** – Sample, pixel

**PSNR** – Peak signal-to-noise ratio

**PU** – See **CTU**.

**QP** – Quantization parameter

**RDO** – Rate-distortion optimization

**RGB** – Red, green and blue, the color space used in computer screens.

**SATD** – Sum of absolute transformed differences

**SSIM** – Structural similarity

**TU** – See **CTU**.

**VLC** – Variable length codes, a form of entropy coding.

**VP9** – See **MPEG-1**.

**YCbCr** – Luma, blue-difference and red-difference, the color space used in video coding.
Appendix II: Detailed results

The left column in the images below contains the bitrate-PSNR graphs for all the tested videos. The right column displays the improvement (bitrate reduction) achieved using MB/CU-tree compared to the other modes of the same encoder. Negative values mean that MB/CU-tree fared worse.