Efficient Color Quantization by Hierarchical Clustering Algorithms

Ville Hautamäki

25.2.2005

University of Joensuu
Department of Computer Science
Master's thesis

Abstract

In color quantization, we create a small palette from the image, and then map pixel by pixel all colors in the image to palette colors. Thus the quantized image contains only colors from the palette. Palette generation is a clustering problem in three-dimensional space. As the clustering problem is NP-complete problem, feasible palette generation algorithms are either heuristic or approximative in nature. Hierarchical methods are a much researched heuristic clustering algorithm class.

There are two basic categories of hierarchical algorithms: *divisive* and *agglomerative*. First we review existing divisive and agglomerative algorithms. Notoriously divisive methods are fast and agglomerative are slow. In this thesis, we present our own agglomerative clustering algorithm that uses k *nearest neighbour graph*. The goal is to design an agglomerative clustering algorithm that runs as fast the divisive methods.

We study experimentally the performance of the nine different clustering algorithms. As testing material we have three images. Test will be run on different palette sizes. We will study, the quality of the quantized and how fast quantization was performed.

Keywords: color quantization, vector quantization, hierarchical clustering, clustering, agglomerative methods, divisive methods, kNN graph

Acknowledgements

First, I would like to give the greatest gratitude and thanks to my supervisor Prof. Pasi Fränti. He not only is a diligent and hard working supervisor, but he also introduced me to this interesting field of algorithms and signal processing with his course of image compression in the autumn 1999.

Also I would like to give thanks to the Prof. Jussi Parkkinen and Prof. Xiaolin Wu for interesting and inspiring discussions and lecture courses on different topics of algorithms and pattern recognition. Thanks goes also to the whole speech processing and image compression research group for helping me to solve different problems and for creating an enjoyable working environment. For financial support, thanks goes to the National Technology Agency of Finland (TEKES).

Big thanks goes to my close friends and colleagues Ismo Kärkkäinen and Tomi Kinnunen. Numerous problems we have "solved" on our whiteboard. Big thanks for helping with matlab and some occasional proofreading.

Big thank you goes to my whole family and all my friends, I would like to point out special thanks to my brother Kalle for helping me out with this thesis. To the whole Bahá'i community of Joensuu I would like to express deep gratitude for friendship and support during those ten years of my life in this city.

And lastly, greatest thanks goes to my wife Rosa, the love of my life, for the infinite support and help during the thesis work.

Contents

Chapter 1

Introduction

"Prediction is very difficult, especially of the future."
– Niels Bohr

The idea of *color quantization* is to reduce the number of colors in a full color (24-bit) image. A smaller set of representative colors is called *color palette*. Reduction should be performed so that the quantized image differs as little as possible from the original image. A 24-bit color image contains at most $2^{24} = 16777216$ different colors. These colors are represented as three dimensional vectors, each vector element having 8-bit dynamic range, allowing $2^8 = 256$ different values. These vectors are often called *RGB triplets*.

In color quantization, the original colors of the image are mapped to a smaller subset of colors. Algorithmic optimization task is to find such a color palette that the overall distortion is minimized. An example of color quantization is depicted in Fig. 1.1. First, a color palette is found by using clustering algorithm and then the original image values are replaced by their closest values in the palette.

11	13	15	17
13	25	27	19
15	27	21	21
17	19	21	23

11	17	23

$\Longrightarrow$

11	11	17	17
11	23	23	17
17	23	23	23
17	17	23	23

Figure 1.1: Example of quantization, where the original image is shown left, the color palette in the middle, and the quantized image in the right.

Fig. 1.2 shows an example of quantizing a 24-bit full color image to 16 colors. We

can see that general features of the image are retained in the quantized image: the two parrots are in the same positions and even small details in the heads of the parrots are mostly untouched by the quantization. On the other hand, a lot of new edges appear in the background. This phenomenon is called a *false contouring*, and it results from not having enough colors to represent smooth change from one color to another. Areas with smooth changes are called *color gradient* or *low frequency* areas in the image. Elimination of the false contouring is one of the most challenging problems in color quantization.

Figure 1.2: 24-bit parrots image quantized to the 16 colors.

Originally color quantization research was motivated by the need to show 24-bit color images in color monitor [50], which were able to handle only 256 different colors. In 1994, Xiang *et al.* wrote: "*In our instructional lab, we have only a few 24-bit workstations, which means that students usually have to create their full-color projects on 8-bit machines*" [107]. This situation has completely changed from 1994. At the moment of writing this thesis, 24-bit display adapters are common, thus negating the need for real-time color quantization in personal computer color displays. Therefore, we need to find another motivation for our work. In the following, we list a few applications, where color quantization can be found.

1.1 Object Recognition and Image Analysis

Content-based image retrieval is a problem, in which we want to find a set of images that match closely to the query image provided by user. A closely related problem is *object recognition*, where one part of the query image needs to be matched against the database of images, for example detecting melanoma cases in medical imaging [66]. In

both problems, color histograms of the database and query images are used as *feature vectors* in the matching. The idea is that image retrieval algorithm computes distance between feature vectors of the query image, and the corresponding database images. Images with the smallest distances are retrieved for the user. If color quantization is used as a pre-processor, quantizing 2^{24} histogram bins to 15 bins, for example, recognition accuracy is increased from 37% to 78% according to [80]. At first glance this result is counter intuitive because information is thrown away. However, quantizing to 15 bins can be considered as a *dimensionality reduction* method that are known to improve recognition accuracy in general [89].

A related application is automatic text location detection from complex color images. Possible images are magazine covers, where automatic process should be able to detect all textual fields [61]. If the algorithm knows, which is the background color, then text location detection is easy. In binary images, this is possible, as one of the colors is background color. In color images, however, the number of potential background colors is higher. This means that a lot of comparisons needs to be done before the background can be separated from the foreground. Color quantization can be used as a preprocessing step for reducing the number of comparisons, and therefore, to make it easier to separate foreground colors from the background colors. Real life color image has multiple colors in background and foreground, especially if the background has a scenery photo and foreground is overlayed text. For that reason, color quantization method has to be careful that some foreground and background colors are not assigned to the same cluster [61].

Caption segmentation from video frames [18] is a specific application of text location detection. The task is to find captions from video frames, which can be further fed into *optical character recognition* (OCR) system. Extracted textual caption can then be used in indexing of digital video libraries [18]. In caption location problem, candidate caption pixels are first extracted using intensity difference method [19]. The underlying assumption is that the caption and the background pixels have a high difference in intensity. Caption areas can then be found from the vertical candidate pixel color groups, and for that reason, the pixel colors are quantized. *Iterative Self-Organizing Data Analysis Technique A* (ISODATA) [5] clustering algorithm has been used in [18]. As the number of caption groups for each video frame is not known in advance, the problem of the number of color clusters needs to be solved also.

1.2 Image Compression

A document image is a *bi-level image* containing text, *continuos tone images* or a combination of those two. Bi-level images are binary images, with only black and white color pixels and continous tone images are images with more color tones and usually areas of gradual change from one color to an other [44]. Traditional image compression methods can be divided in two different categories: bi-level and continuous tone, depending on the type of image to be compressed. Context based compression schemes, such as JBIG2 [51] can compress efficiently drawings with a low number of colors, and scanned text. However, JBIG2 does not work well on continuous tone images such as digital photographs, and compression to 6.23 bpp (out of 8 bpp) can be achieved, on average [36].

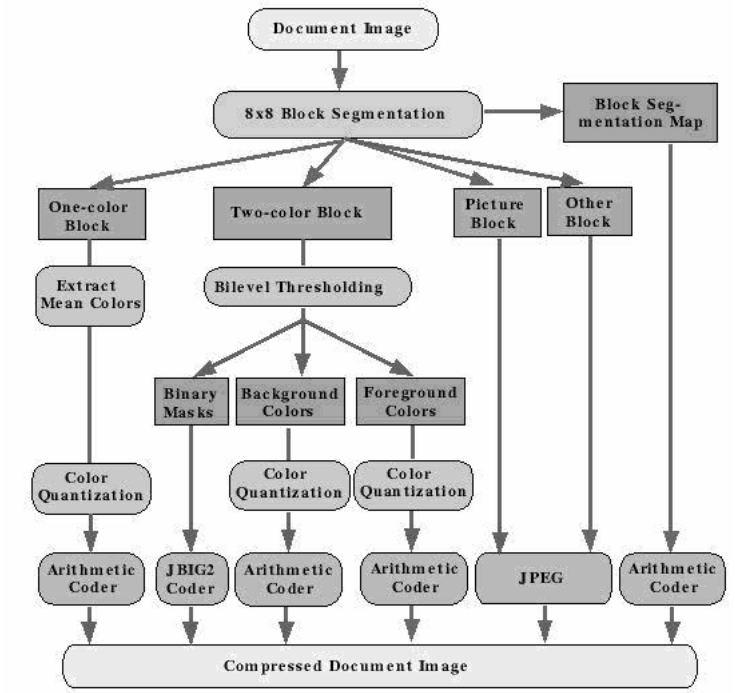


Figure 1.3: A document compression system using color quantization [17].

For digital photographs, a better approach is to apply lossy compression such as JPEG [78] or JPEG2000 [87]. On the other hand, drawings and text usually contain only sharp edges, and those are difficult to compress without noticeable visual degradation by JPEG. One possible approach is to divide image into 8x8 pixel blocks, and classify each block to three fundamental categories: *background blocks*, *bi-level*

blocks and *continuous tone blocks*. An example of this process is shown in Fig. 1.3. Color quantization is used for finding foreground and background colors in the bi-level blocks [17].

Mobile devices usually do not have full color displays. We will refer these displays to *low end displays*. When viewing full color images on such a device, color quantization is needed. In [36], the following scheme has been proposed. First, a full color aerial image is lossy compressed and then transmitted over the network in compressed form. In mobile phone, the aerial image is then color quantized for display. This scheme produced less distortion for a given rate than performing color quantization on aerial image before the compression and transmission over the network. In Fig. 1.4, the upper part describes a scheme where color quantization is performed in the mobile device and the lower part describes a scheme where color quantization is performed beforehand in the server.

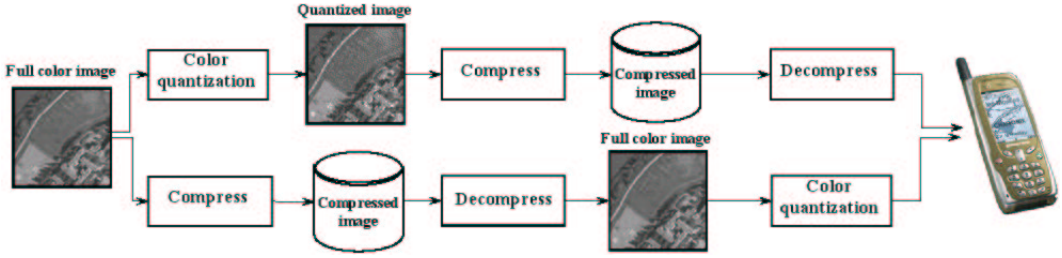


Figure 1.4: System diagram of using color quantization and compression in mobile device.

1.3 Other Applications

In computer graphics, a relatively new research area is *non-photorealistic rendering* (NPR). The goal of NPR is to render image "in more creative fashion", for example rendering 3-d scenery or models in water color painting style instead of photorealistic rendering [25]. Yeh *et al.* [110] used color quantization as a shading technique in their work on rendering 3-d models of animals in traditional Chinese painting style. To achieve painting style on shading, *Gouraud shaded 3-d models* [35] have been quantized to four colors.

Map images typically contain only few colors, and even though these images should not be compressed with JPEG or similar lossy compressiom mehtods, users still distribute JPEG compressed map images in internet [63]. Unfortunately, JPEG

compression of map images creates so-called *ringing* around the edges due to the quantization of the cosine function coefficients [76]. As edges are discontinuities, DCT transform uses more coefficients than are needed for smoothly changing areas in the image, which is the case for continuous tone images [44]. Similar problems arise when using other lossy compression techniques. Color quantization can be used to suppress ringing noise in JPEG compressed map images as shown in Fig. 1.5 [63].

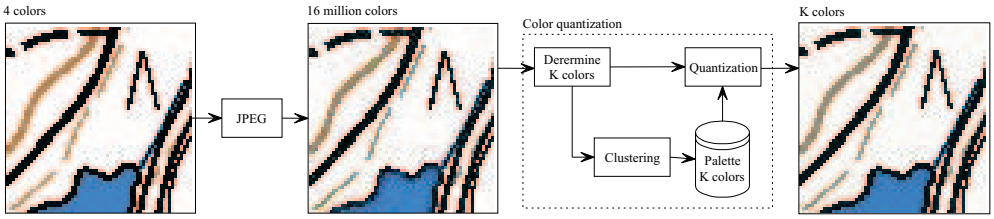


Figure 1.5: Recompression of map images needs color quantization [63].

1.4 Goals of This Study

In this thesis, we formulate color quantization as an optimization problem. The goal is to find the optimal color palette for a given image so that when the image is quantized using this palette, the quantization error is minimized. When considering the original color distribution of the image, finding the optimal palette is usually formuleted as finding optimal *partitioning* of the color space. In the pattern recognition literature, term *cluster* is used instead of partition [89], and algorithms that find the clusters are called *clustering algorithms*. Finding the optimal palette was shown to be NP-complete by Garey *et. al.* [73]. In this work, the authors found a polynomial time reduction of the *exact covering by three-sets* problem to the restricted version of the quantization decision problem. For this reason, heuristic or approximative solutions are needed.

Two main approaches have been used in the literature: (i) *iterative methods* and (ii) *hierarchical methods*. Iterative methods start with an initial partition, which is usually a random solution, and in each iteration the algorithm changes the partitioning towards a solution with less distortion. The most widely known iterative method is *Generalized Lloyd Algorithm* (GLA) [70], which is also known as *K-means algorithm* in the pattern recognition literature. Hierarchical methods work either in *top-down* fashion, splitting the color space into smaller and smaller partitions, or in *bottom-up* fashion, enlarging partitions one at the time.

In this thesis, we concentrate on hierarchical methods for palette generation. On the other hand, most common iterative and heuristic color quantization methods are also reviewed. Among the hierarchical methods we focus on the bottom-up methods, which are called *agglomeration methods*. Agglomerative clustering algorithm in its basic form is easy to implement and it usually produces good results [94]. We first initialize the system by assigning all unique colors to their own clusters. After initialization, agglomeration iteratively merges the closest pair of clusters until the number of clusters matches the user defined palette size. A simple way to find the closest pair of clusters requires enumeration of all pairs. One way to speed up the agglomeration is to precalculate a few (k) nearest neighbours to all colors, and to search the closest pair by considering only these neighbours. The clusters together with the links to their k closest neighbours forms a k *nearest neighbour graph* [32]. An important research question is how big k should be so that the quality of the generated palette is still close or equal to the agglomerative clustering in its basic form.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. First color quantization problem is defined and a few commonly used heuristic algorithms are described in section 2. Divisive methods are reviewed in section 3. Agglomerative methods are described in section 4. Graph based agglomerative clustering method is introduced in section 4.3. Selected algorithms are tested exerimentally in section 5. Conclusions are drawn in section 6.

Chapter 2

Color Quantization Framework

"General rules apply, except when they don't"
– John McCarthy

Color quantization consists of two main tasks:

1. Palette generation
2. Mapping the original colors to the palette colors.

An input image $I(x, y)$ contains $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\} \subset \mathbb{N}^3$ unique colors. The parrots picture in Fig. 1.2 is an image of size 768x512, which yields to 393216 pixels, of which 72175 are unique. A palette generation algorithm finds a smaller represenative set of colors $C = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots \boldsymbol{c}_M\} \subset \mathbb{N}^3$ from the N distinct colors. The second task is to map the colors in the original image I one pixel at the time to a palette color $c_i \in C$. This process of color mapping produces the output image $\hat{I}(x, y)$.

In Fig. 2.1, a schematic diagram of the color quantization system is shown. To reduce the computational load of the palette generation algorithm, it is sometimes useful to first prequantize the input color space to a smaller number of colors.

Figure 2.1: Description of the color quantization framework.

In color quantization the goal is to find such a C that minimizes *distortion* $D(I, \hat{I})$ between original I and mapping $q : X \to C$. Intuitively we would like distortion

function $D(\cdot,\cdot)$ to be such that if visual distortion between images I and $\hat{I}$ is large, then $D(I,\hat{I})$ should also be large. We also require that if $I=\hat{I}$, then $D(I,\hat{I})=0$. It is now easy to see that two decisions affect the final outcome, namely how color mapping is done and how the palette colors are selected. Both of these decisions depend on the selected distortion function.

In section 2.1, we consider the color space that we will use in this thesis. We define scalar quantization problem in section 2.2. We generalize this definition to vector quantization, and thus, also to color quantization in section 2.3. Section 2.4 is dedicated to the uniform quantization, and then we study different methods for pre-quantization in section 2.5. In section 2.6, we explain how distortion metric can be defined so that it leads to visually more pleasing outcome. In section 2.7, we study algorithms to generate the color palette.

2.1 Color Spaces

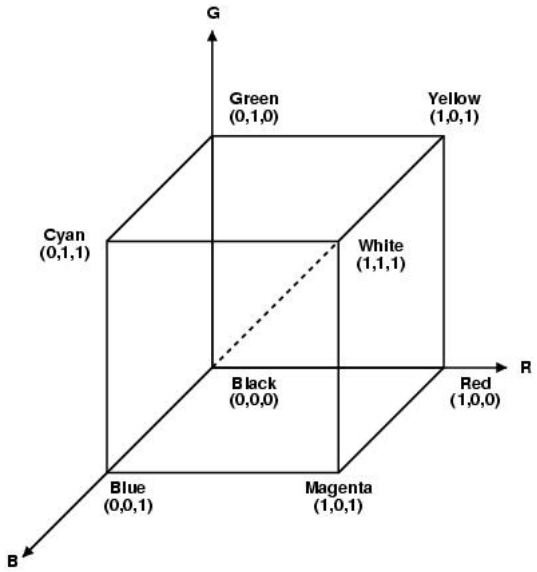


Figure 2.2: RGB color cube [44].

In this thesis, we concentrate our studies in *tristimulus* color spaces, which means that each color is represented by a three component vector ($\boldsymbol{x}=(R,G,B)$) in RGB color space as shown in Fig. 2.2. The corners of the RGB color cube represent the colors red, green, blue, cyan, magenta, yellow, black and white. Black is the origin

and white is the opposite corner farthest from the origin. In this model, different shades of gray reside in the main diagonal from the (0,0,0) to the point (1,1,1). The component color values are in range [0,255]. For the ease of interpretation all color values have been normalized to the range [0,1] in Fig. 2.2.

The choice of the tristimulus color space is based on the fact that color vision in *human visual system* (HVS) is based on three different types of cones (namely L, M and S). Wavelength response curves of those cones are shown in Fig. 2.3. From these response curves, standard *color matching functions* for XYZ color space are defined as $x(\lambda)$, $y(\lambda)$ and $z(\lambda)$, where λ is the wavelenght of the observed color [20].

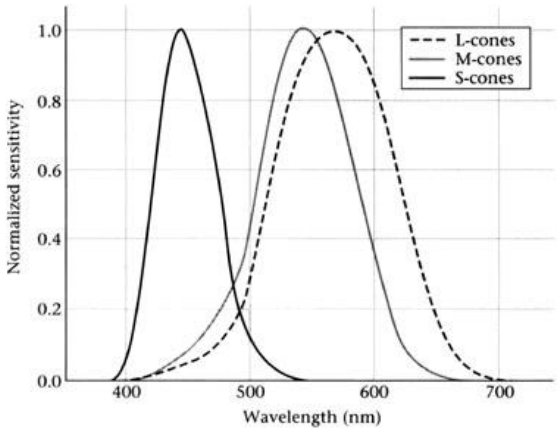


Figure 2.3: Wavelength response curves of L, M and S cones [98].

A color in the XYZ space can be transformed to another three component color space, for example RGB, CIELAB and YIQ [98]. A problem with the XYZ space and the RGB derived from it is that the space is not *perceptually uniform*, which means that distance in the color space between two colors does not match the perceived difference. On the other hand, CIELAB 94 color space standard is designed to be perpectually uniform. This is the reason why some color quantization systems transform RGB vectors first to CIELAB 94 space and then perform palette generation in this space [112].

Even though CIELAB 94 space is perceptually uniform, most of the images to be quantized are stored either in RGB or in YIQ space [44]. For instance, in MPEG and JPEG compression standards, a variant of YIQ color space is used. The YIQ color space has also been proposed to be used as color space, when color quantization should be performed [102]. Transformation from one tristimulus color space to the CIELAB

increases the computational load of the color quantization system. Furthermore, transforming RGB images into CIELAB space before quantization causes rounding errors, which can be seen in the final quantization results [112].

2.2 Scalar Quantization

In this thesis, we define the *scalar quantization* [46] as a process of reducing the number of levels in the original discrete signal.When using scalar quantization, we quantize a set of integers, which form a *histogram*. Scalar quantization process produces a smaller set of numbers, where the original histogram is mapped to. More formally, quantization is the process of mapping all original values $x \in \mathbb{N}$ in the same *quantization cell*, or *partition* S_i, to the same *reproduction value* $c_i \in \mathbb{N}$. The input set X is divided into partitions S_i so that $S_i \cap S_j = \emptyset$, when $j \neq j$ and $\bigcup_{i=1}^{M} S_i = X$. Reproduction values are usually called *reproduction levels*. A set $C = \{c_1, c_2, \ldots, c_M\} \subset \mathbb{N}$ contains all reproduction levels of the quantization system. Quantization is achieved through mapping $q : \mathbb{N} \rightarrow C$, called quantization:

$$c_i = q(x). \tag{2.1}$$

The quantization rule can be rewritten shortly as [46]:

$$q(x) = \sum_i c_i 1_{s_i}(x), \tag{2.2}$$

where $1_{s_i}(x)$ is an indicator function giving value 1 if $x \in S_i$ and 0 otherwise. The partitions and the reproduction values are closely linked together. By definition there exists only one reproduction value per partition. It is important to notice that the partitions are disjoint, which means that each input value belongs only to one partition. Unfortunately, the $q(x)$ is not a bijection, and thus, inversion is not possible in general. In other words, for a given output level, we cannot retrieve the input value x.

An example of quantization is shown in Fig. 2.4, where real numbers are quantized to natural numbers. This familiar quantization rule is rounding off the decimal numbers to the nearest integer. We can also calculate the maximum *error* between the input value x and the $q(x)$ in this case. Maximum error appears, when the input value is exactly in the half way between two integers, and it then amounts to $|x - q(x)| = 0.5$.

11.3298	13.04	15.62	17.343	$\Longrightarrow$	11	13	16	17

Figure 2.4: Example of scalar quantization, where $q : \mathbb{R} \to \mathbb{N}$.

Rounding off is an example of a *uniform* quantizer [46], in which the quantization levels c_i are equispaced (say Δ apart).

Keeping in mind that the goal of the quantization is to produce quantization levels with the minimum error, we need to be concrete on what error function we are minimizing. In quantization literature, the error is usually called *distortion*, and will be denoted here as $d(x, q(x))$ [46]. In the previous example, distortion type is called *absolute error*, which is an intuitive distortion measure. More often the *squared error* is used as a distortion function:

$$d(x, q(x)) = |x - q(x)|^2. \tag{2.3}$$

Rounding the real number to its closest integer, squared error distortion still gives maximal error to be the middle point between two integers. Error in this case is $(x - q(x))^2 = 0.25$.

Our task is to quantize the one dimensional input (in the case of scalar quantization) to M quantization levels. To define the problem formally [46, 100], let $X = \{x_1, x_2, \ldots, x_N\}$ be the values of the input set and $x \in \mathbb{R}$. Then the number of times each input value x_i appears are $P = \{p_1, p_2, \ldots, p_N\}$. From now on we will call this quantity a *frequency*. The quality of the overall quantization system can then be expressed as the *mean squared error* (MSE):

$$\frac{1}{\sum_{i=1}^{N} p_i} \sum_{i=1}^{M} \sum_{x_j \in S_i} p_j d(x_j, c_i). \tag{2.4}$$

The term $\sum_{x_j \in S_i} p_j d(x_j, c_i)$ is the distortion incurred from mapping all values in partition S_i to c_i. Distortions from all the partitions are then summed and divided by the total frequency. The objective of quantization task is to find such set $C = \{c_1, c_2, \ldots, c_M\}$ that minimizes Eq. (2.4). When we generate one set C, we say that we have generated a *solution*.

In scalar quantization, we have one dimensional space, in which ordering exists between all values in the following way:

$$-\infty < x_1 < x_2 < \ldots < x_{N-1} < x_N < \infty. \tag{2.5}$$

In a given solution, if one code cell includes values x_1 and x_2, then x_4 cannot be added to this cell without adding the value x_3, because non-convex quantizer cells are not allowed [41]. For this reason, the optimization problem can be split the into subproblems. Optimal solution to the scalar quantization problem can be found by *dynamic programming* [6], where the idea is to build the globally optimal solution to a problem in bottom-up fashion by solving subproblems and storing the solved solutions. After one set of solutions is done, those solutions can be used to solve one set bigger solution and so on. The first one to use dynamic programming to solve scalar quantization problem was Bruce [11]. The time complexity of his solution is $O(MN^2)$. Later Wu [100] improved this result to the $O(MN)$ by using the monotonicities of the l_2 error distortion function. In 1993, Wu and Zhang generalized that result to squared error distortion function [105].

2.3 Vector Quantization

Vector quantization (VQ) [41, 46, 70] is a generalization of 1-dimensional scalar quantization to any dimension D. In color quantization, we are dealing with three dimensional RGB vectors, and thus, color quantization is a special case of vector quantization where $D = 3$.

It is typical for a vector quantization algorithm to employ a large amount of nearest neighbour searches during the process of the algorithm [94]. Some color quantization algorithms use a special search structures that utilize the fact that the data is three dimensional to speed up the nearest neighbour searches [42, 107]. In this work, however, we concentrate on algorithms that can be used also in the general case where D is not restricted.

Definitions needed to formalize the vector quantization problem are the same as in scalar quantization, except that the quantizer input and output are vectors. Let the input vectors be the set $X = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\} \subset \mathbb{N}^D$. Quantizer cells are defined as in scalar quantization, where quantizer cells S_i partition the input space into disjoint partitions $S = \{S_1, S_2, \ldots, S_M\}$. Each partition is associated a *reproduction vector* $\boldsymbol{c}_i \in \mathbb{N}^D$, which is drawn from the set $C = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_M\} \subset \mathbb{N}^D$, called the *codebook*. The quantizer rule can be rewritten as:

$$\boldsymbol{c}_i = q(\boldsymbol{x}). \tag{2.6}$$

Using the quantization rule, we now define the partition formally:

$$S_i = \{\boldsymbol{x} \in X |\ q(\boldsymbol{x}) = \boldsymbol{c}_i\}. \tag{2.7}$$

We can also define the partitioning using the inverse of the quantization rule as $S_i = q^{-1}(\boldsymbol{c}_i)$ [92].

An example of vector quantization is shown in Fig. 2.5, where the vector $\boldsymbol{x}$ is quantized to the reproduction vector $\boldsymbol{c}_1$. The quantization rule will assing every vector in the partition S_1 to the reproduction vector $\boldsymbol{c}_1$. It can be seen that the vector $\boldsymbol{x} \in S_1$ will be quantized to $\boldsymbol{c}_1 = q(\boldsymbol{x})$. In a similar fashion, all vectors in the partition S_2 will be quantized to the codevector $\boldsymbol{c}_2$.

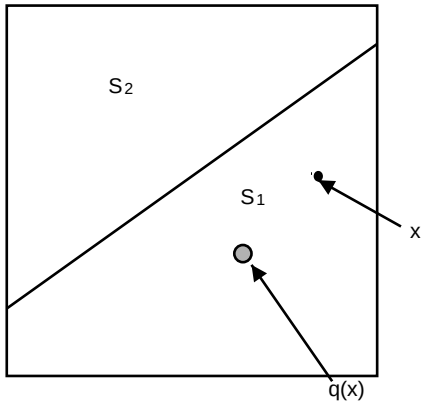


Figure 2.5: Example of vector quantization.

Squared error in vector quantization is expressed as:

$$d(\boldsymbol{x}, q(\boldsymbol{x})) = ||\boldsymbol{x} - q(\boldsymbol{x})||^2 = (\boldsymbol{x} - q(\boldsymbol{x}))^t(\boldsymbol{x} - q(\boldsymbol{x})), \tag{2.8}$$

where t denotes transpose. From now on, we will assume $d(\cdot, \cdot)$ to be squared error distortion function if not stated otherwise. This does not prevent us using some other distortion function. In section 2.6, we will see that if we want to change the distortion function to reflect better the percieved error, it will also change our optimization problem.

Attractiveness of the squared error is its simplicity and the fact that its theoretical properties are well known [46]. The term $||\boldsymbol{x} - \boldsymbol{c}||$ in the squared error is called l_2 norm or *Euclidean distance*. The l_2 *norm* is defined as [41]:

$$||\boldsymbol{x} - \boldsymbol{c}||_2 = \Big(\sum_{i=1}^{D} |x_i - c_i|^2\Big)^{1/2}. \tag{2.9}$$

In fact, this is a special case of a more general distortion metric, called *Minkowski metric* or l_p*-norm*:

$$||\boldsymbol{x} - \boldsymbol{c}||_p = \Big(\sum_{i=1}^{D} |x_i - c_i|^p \Big)^{1/p} \tag{2.10}$$

If we set $p = 1$ we get l_1, namely the *Manhattan* or *city block metric*. Similarly taking the limit $\lim_{p\to\infty}$ we get l_∞, which is also called the *maximum metric*.

We can now use the distortion metric to define the partition by:

$$S_i = \{\boldsymbol{x} \in X | d(\boldsymbol{x}, \boldsymbol{c}_i) \leq d(\boldsymbol{x}, \boldsymbol{c}_j) \quad \forall j \neq i\} \tag{2.11}$$

By combining the Equations (2.4) and (2.8), we get the objective goal for the vector quantization as:

$$C = \arg \min_{\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_M\} \subset \mathbb{R}^D} \frac{1}{\sum_{i=1}^{N} p_i} \sum_{i=1}^{M} \sum_{\boldsymbol{x}_j \in S_i} p_j ||\boldsymbol{x}_j - \boldsymbol{c}_i||^2 \tag{2.12}$$

In other words, the goal is to find such a codebook C that minimizes squared error distortion after quantizer mapping. Consequently this is also the task of the codebook generation in the color quantization. Only unknown variables in Eq. (2.12) are the set of codevectors C, as partitioning is defined by Eq. (2.11).

Another useful distortion metric is *total squared error* (TSE), which can be obtained from Eq. (2.12) in the following way:

$$\mathrm{TSE}(X, C) = \sum_{i=1}^{M} \sum_{\boldsymbol{x}_j \in S_i} p_j ||\boldsymbol{x}_j - \boldsymbol{c}_i||^2. \tag{2.13}$$

An obvious algorithm to find the optimal solution is to try every possible codebook and pick the one that minimizes distortion. In color quantization, all possible codevectors are drawn from RGB color space with 2^{24} different possible codevectors. The number of all possible codebooks can be expressed combinatorially:

$$\begin{aligned} \binom{2^{24}}{M} &= \frac{2^{24}!}{(2^{24} - M)!M!} \leq \frac{2^{24}!}{(2^{24} - M)!} \\ &= (2^{24} - (M-1))(2^{24} - (M-2)) \cdots 2^{24} \leq 2^{(M-1)24}. \end{aligned} \tag{2.14}$$

With this analysis we can see that enumerating all possibilities is not a sensible approach.

The color quantization task is a *combinatorial optimization* task, which has been proven to be *NP-complete* [73]. This is in stark contrast to the scalar quantization problem, which was show to be solvable in *linear* time [100].

In Fig. 2.6, there are 4000 randomly sampled pixel values from *Parrots* image (Fig. 1.2), plotted according to their green and blue dimensions. On the right hand side, the open circles represent the 16 code vectors and the lines represent the partition borders. These regions are called *Voronoi partitions* [27].

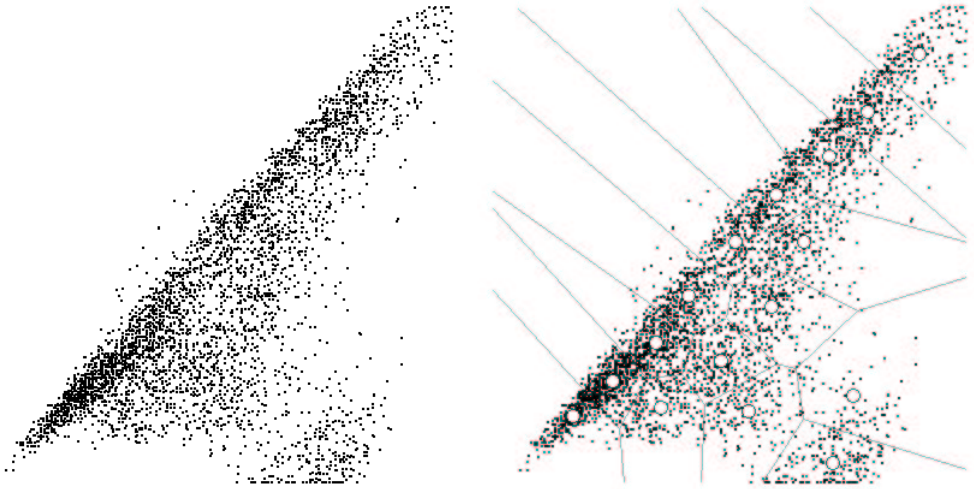


Figure 2.6: Example of vector quantization: original dataset is shown to left, and quantized to 16 codevectors to right.

Lloyd found in 1957 [72] the *necessary conditions* of optimality for a reproduction levels for the squared error distortion. The conditions were later extended to vector quantization by Gersho and Gray [41]:

- *Nearest neighbour condition*: For a given input vector, the optimal reproduction codevector is its nearest neighbour in the codebook. From Eq. (2.11) we get the condition:

$$d(\boldsymbol{x}, q(\boldsymbol{x})) = \min_{\boldsymbol{c}_i \in C} d(\boldsymbol{x}, \boldsymbol{c}_i). \tag{2.15}$$

- *Centroid condition*: For a given partition, the optimal code vector is the centroid

(average vector) of the vectors within the partition:

$$\boldsymbol{c}_i = \frac{1}{|S_i|} \sum_{i=1}^{|S_i|} \boldsymbol{x}_i. \tag{2.16}$$

First condition can be interpreted as the optimality condition for partitioning the dataset, given the centroids. Unfortunately these conditions are not *sufficient* for the globally optimal codebook [41].

We saw that the palette generation problem in color quantization is NP-complete. This leads to the fact that the only feasible algorithms are heuristic ones. We can compare different heuristic algorithms by the quality of the generated palette, as a comparison metric we use MSE. Not only is the distortion of the solution important, but the time complexity is also important to keep in mind. For this reason, we want to find a heuristic algorithm that will be as good as possible in time-distortion sense. Evaluating the goodness of the algorithm is usually done empirically.

2.4 Uniform Vector Quantization

Uniform vector quantizer is a VQ where codevectors $\boldsymbol{c}_i$ are drawn from a *lattice* [21]. Geometrically, lattice can be thought of as an infinite regular array of points, which uniformly fills the D-dimensional space [91]. As an example $Z_n = \{\boldsymbol{x} \in \mathbb{Z}^n\}$ is an n-dimensional lattice, where *lattice points* are all vectors in n-dimensional space, where vector components are integers. Lattice is inherently an infinite in size, as seen from Z_n, thus for color quantization, we need to *truncate* it to the size M.

In Fig. 2.7, an example of uniform quantizer with rectangular and hexagonal lattices are given. Note that as the lattice is an inherently infinite structure, the quantization cells are not fully square or hexagonal at the edges. It is interesting to note, that hexagonal lattice is much better for this dataset than square one, because it gives smaller MSE even with fewer codevectors. It is also clear from the figure that uniform quantization cannot give optimal C, except in a some special cases. The reason for this is that the input vectors are not usually uniformly distributed. Typical situation is that one region of the vector space has input vectors in dense clusters, and in other regions, vectors are sparsely distributed.

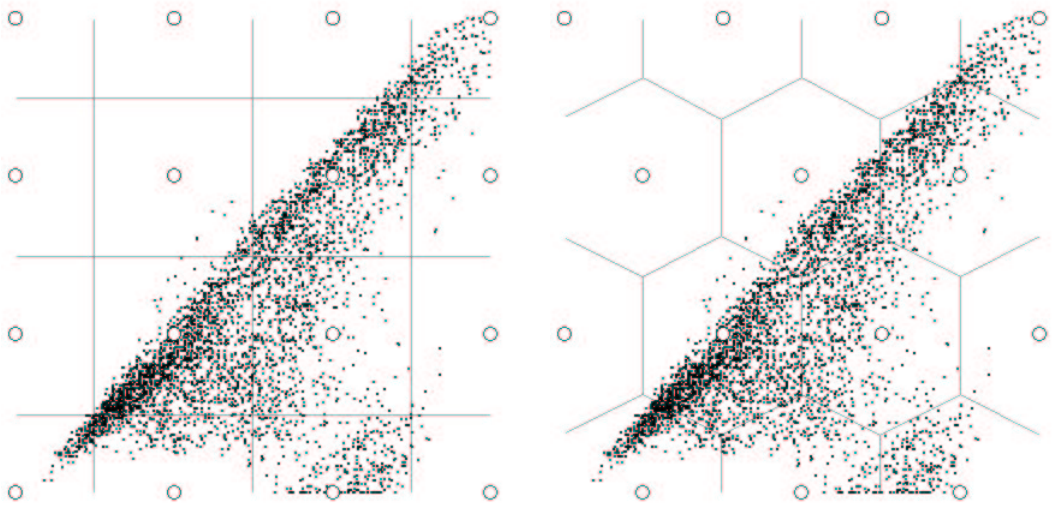


Figure 2.7: Example of uniform vector quantization to 16 codevectors (MSE = 635.13) and to 14 codevectors (MSE = 593.31).

2.5 Prequantization of the Dataset

Dataset size in color quantization is usually very large. Potentially full color image can contain up to 16 million different color vectors, even though this kind of worst case image is very rare in practice. Nevertheles, images with even moderate size can have a large number of unique colors and, thus, making the size of the dataset (N) very large and affecting the running time of the algorithm adversely. For example, parrots is an 768x512 image, which yields to 393216 pixels, of which 72175 are unique. One example of palette generation algorithm have time the time complexity of $O(N^2)$. For that algorithm to generate palette would require more than five billion steps. This is the reason why many color quantization algorithms [50, 102, 92, 107] use some form of prequantization as a preprocessing step.

Prequantization is the process of reducing the size of the dataset so that the palette generation would be faster. With prequantization proposed in [50], 72175 vectors was brought down to around 8000 vectors. In this case, an $O(N^2)$ algorithm would need around 64 million steps with this prequantization scheme.

Formally, prequantization is a mapping $p : X \rightarrow X'$, so that the original dataset is reduced to the size $|X'| = N'$, where $N > N' > M > 0$. In general, we can divide

prequantization methods into three categories:

1. *Bitcutting* is the process of cutting the number of least significant bits from R, G and B components. This is effectively the same as uniform quantization of RGB color space into N' possible colors [81].

2. *Subsampling* is the process of selecting a subset of colors from all the possible unique colors in the original image.

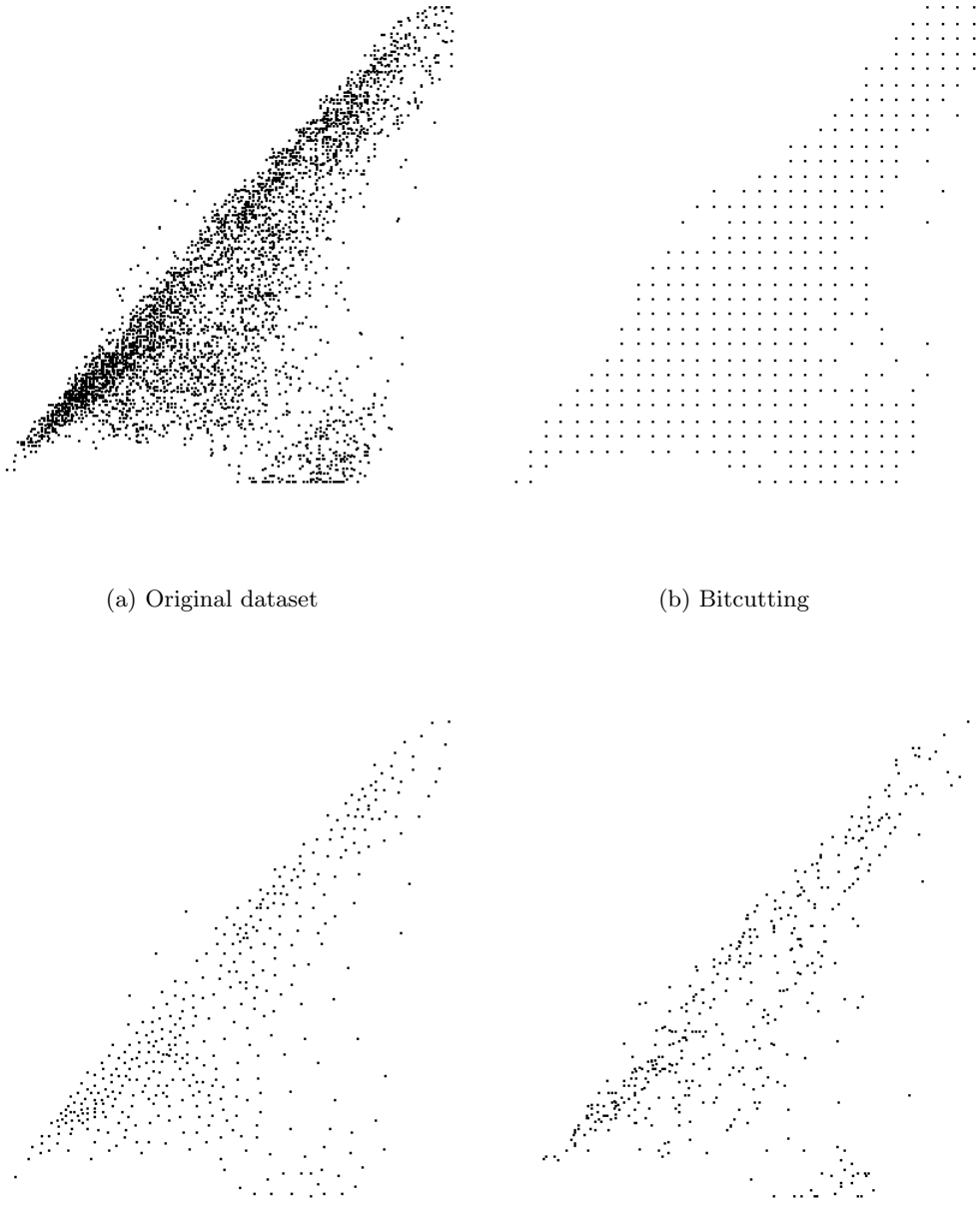
3. *Clustering* the original dataset X to N' by any fast clustering algorithm.

In Fig. 2.8, we see an example of different prequantization methods, when the dataset of size 4000 was reduced to 400 vectors. When comparing prequantized set to the original, we see that bitcutting does not represent dataset the accurately. In the bitcutting method, we reduced the resolution per vector component from 8bit to 5bit. Both random sampling and clustering approach produce fairly representative subset of the original dataset. Visual inspection shows that clustered dataset seem to be distributed more evenly than the randomly sampled version.

Most of the color quantization methods use bitcutting as the prequantization method [50, 69, 107]. First suggestions to use prequantization came from Heckbert [50], where he suggested cutting 3 least significant bits off from each R, G and B component. In his method, precision was reduced from 24bits per color to 15bits per color. Later many color quantization algorithms fix precision to 15bits ($R_5 + G_5 + B_5$; $R_5 + G_6 + B_4$) or 18bits ($R_6 + G_6 + B_6$; $R_6 + G_7 + B_5$) per color [81]. However, Shufelt showed that working with less than 21bits can noticeably impair the quality of the quantized images [84] .

The assumption in the subsampling is that not all colors are necessary to build final codebook. Sampling should be done in the image domain, when reading the image into memory, for example, random sampling the fixed number of pixels [28]. Another possibility is sampling the image at decreasing step sizes (*e.g.* $1009, 757, 499, 421, \ldots$) [93], or to sample every nth pixel in every mth row [81].

The time complexity of both bitcutting and subsampling is linear, and not every pixel is needed to process. However, duplicate colors might get to be sampled in. To keep the dataset unique for the clustering algorithm we need to store the prequantized colors in the binary tree, thus the time complexity yields to $O(N \log N)$. Every color is inserted to the tree one by one, and if a color in question is already in the tree then the frequency of the color is increased by one.



(a) Original dataset

(b) Bitcutting

(c) With Clustering

(d) Random sampling

Figure 2.8: Example of prequantization.

Reitan [81], on the other hand, divides prequantization methods into bitcutting and subsampling methods. Contrary to that statement any fast clustering algorithm can also be used to generate smaller dataset. For example *popularity algorithm* [50], can also be used as a prequantization step. Also *generalized Lloyd algorithm* (GLA) [70] has been used to generate a smaller dataset for a slower $O(N^2)$ algorithm [95].

2.6 Perceptual Color Quantization

Human visual system (HVS) is more sensitive to the distortion in low spatial frequancy than in high spatial frequency areas. This means that in the areas of image with low *activity* (i.e. low spatial frequency) eye detects distortions easier than in high activity areas [23]. For example *false contouring* in low frequency areas such as sky are noticed easier than distortion near edges. This effect is called *spatial visual masking* [23].

The intuitive idea to solve this problem would be to move the quantization error from low activity areas to high activity areas where distortion does not matter much. Basically four strategies exist to solve this problem:

1. Allocate more quantization levels to the areas of color space that correspond to the low spatial activity [112].
2. Perceptually weighted distortion metric as a cost function [79].
3. A method that tries iteratively reduce false contouring after pixel mapping phase [56].
4. *Dithering* approach in color mapping instead of mapping to the nearest neighbour palette color [67].

Dithering has been used in color quantization literature [50, 101] as a last stage of color quantization to reduce false contouring. In printing industry, dithering is known as *digital halftoning* [44] where task is to print gray scale image by using only black and white dots. Most common dithering methods are called *error diffusion*, where the idea is to spread the quantization error to neighbouring pixels. In *Floyd-Steinberg* error diffusion [34], quantization error is diffused to those pixels that have not yet been quantized. Image is quantized from left to right and from top to bottom. Fig. 2.9 shows the Floyd-Steinberg error diffusion filter, where X denotes the current pixel.

The quantization error from X is added to the neighbouring pixels according to the diffusion filter so that 7/16 of the error is added to the pixel on right and so on. To a human observer, dithered images appear to contain less error than non-dithered, which is explained as the error is spatially dispersed by adding high frequency noise to make it less perceptible [81].

	X	7/16
3/16	5/16	1/16

Figure 2.9: Floyd-Steinberg error diffusion filter.

Dithering has several problems:

- dithering is a nonlinear process and as such difficult to model mathematically,
- dithered images are hard to compress efficiently using dictionary based compression algorithms [36],
- dithering contradicts the optimization goal of color quantization.

Our assumption in this section is that false contouring can be reduced by using spatial activity weighting. One approach to calculate the spatial activity of the pixel $I(x, y)$ can be performed by calculating the luminance gradient [81]:

$$a(x, y) = |Y(I(x, y)) - Y(I(x - 1, y))| + |Y(I(x, y)) - Y(I(x, y + 1))|, \tag{2.17}$$

where $Y()$ is the luminance of the pixel $I(x, y)$, and $a(x, y)$ is the activity matrix containing activity values for each pixel in image I. Luminance $Y()$ is defined as the Y component from the YIQ color space, where conversion is perfomed using the following formula [44]:

$$Y = 0.299R + 0.587G + 0.114G. \tag{2.18}$$

Then the activity weighting function $w(x, y)$ is defined as [81]:

$$w(x, y) = \begin{cases} 1/4 & \text{if} \quad a(x, y) = 0 \\ 1/3 & \text{if} \quad a(x, y) = 1 \\ 1/a(x, y) & 1 < a(x, y) < 12 \\ 1/a(x, y)^{1.25} & 12 \leq a(a, x) \leq 16 \\ 1/16^{1.25} & \text{if} \quad a(x, y) > 16 \end{cases} \tag{2.19}$$

Cut-off values and the exponent 1.25, was found by trial and error method [81]. MSE distortion function from Eq. (2.12) can be rewritten in terms of distortion between the original image I and the quantized image $\hat{I}$:

$$D_{\mathrm{MSE}}(I, \hat{I}) = \frac{1}{WH} \sum_{x=1}^{W} \sum_{y=1}^{H} ||I(x, y) - \hat{I}(x, y)||^2, \tag{2.20}$$

where W is the width of the image, H is the height and function $I(x, y)$ gives the pixel color from the position x, y. Then weighted MSE is defined as:

$$D_{\mathrm{WMSE}}(I, \hat{I}) = \frac{1}{\sum_{x=1}^{W} \sum_{y=1}^{H} w(x, y)} \sum_{x=1}^{W} \sum_{y=1}^{H} w(x, y) ||I(x, y), \hat{I}(x, y)||^2 \tag{2.21}$$

There are different ways to define the spatially weighted distortion function, for example, to divide image into disjoint square blocks (8 by 8 or 4 by 4 blocks). The spatial activity of the block is calculated by taking the average distance from all color vectors of the block to the mean color in the block [15, 60].

2.7 Palette Generation

In this section, we review some non-hierarchical clustering methods used in palette generation. In general, any clustering method is suitable for palette generation task. The difference between VQ and clustering is that in VQ we try to minimize MSE cost function and in clustering the goal is to find the "natural" clusters from the data [53]. What is natural cluster depends on the data in question, and thus, optimization based clustering methods need to change the cost function accordingly. In palette generation, when squared error distortion is used, cluster shape is fixed to spherical shape. The centroid of the partition is optimal codevector of the partition [46].

2.7.1 Product Vector Quantizer

Scalar quantizer can also be used to quantize vectors in such a way that each vector component is quantized separately. Scalar quantizer for vectors is called *product quantizer* [46]. For vectors $\boldsymbol{x} = (x_1, x_2, \ldots, x_D)$, product quantizer needs to build $C_1, C_2, \ldots, C_D$ codebooks. Final codebook is then the cartesian product of the scalar codebooks C_i:

$$C = C_1 \times C_2 \times \ldots \times C_D. \tag{2.22}$$

The corresponding codebook sizes are $M_1, M_2, \ldots, M_D$. Note, that the final vector quantization codebook size is given by the:

$$M = \prod_{i=1}^{D} M_i. \tag{2.23}$$

Time complexity of the codebook generation using product quantizer is $O(\sum_{i=1}^{D} M_i N) = O(MN)$.

Method, where scalar quantizers are designed independetly of each other is called *independent scalar quantization* (ISQ). Drawback in independet scalar quantization is that it does not take into account correlation between the vector components [4].

In Fig. 2.10 we see example of product quantizer with two optimal scalar quantizers. For both dimensions R and G, optimal scalar quantizer was used to generate 4 codecells, which leads to a vector quantizer codebook of size 16. Note that the upper left corner has three codecells without any vectors from original dataset. We can see that the dataset, where majority of vectors reside in the diagonal is especially difficult for independent scalar quantizer. One option to remedy the situation would be to rotate all vectors before quantization and then rotate vectors back after color mapping.

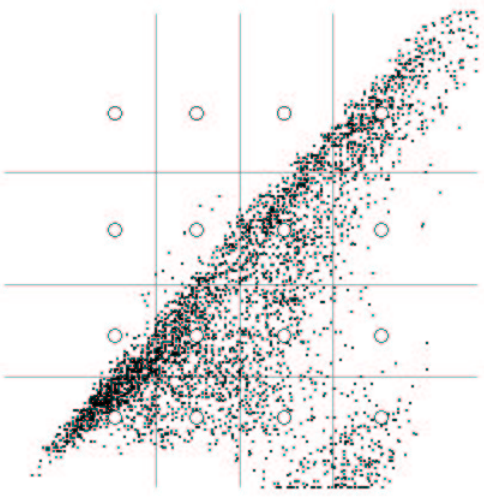


Figure 2.10: Example of product quantizer with optimal scalar quantizer on two dimensions (MSE = 246.51).

Strategy that every dimension has the equal number codevectors is not necessarily

the optimal choice. Akimov *et al.* [1] proposed an algorithm to choose optimally codebook sizes for each dimension and partition.

2.7.2 Sequential Scalar Quantization (SSQ)

In *sequential scalar quantization* (SSQ) [4], scalar components are quantized one at the time taking into account those components that had been quantized already. We denote the ith scalar component of all vectors in the dataset by X_i. The SSQ algorithm is [4]:

1. Qauntize the first scalar component X_1 independetly to M_1 levels, which forms M_1 codecells S_{1j}, $1 \leq j \leq M_1$.
2. Quantize X_2 within each S_{1j} according to its conditional distribution $p(x_2|S_{1j})$ to M_2 regions in total. We call these regions codecells S_{2j}, $1 \leq j \leq M_2$.
3. Quantize X_3 within each S_{2j} according to its conditional distribution $p(x_3|S_{2j})$ to to $M_3 = M$ regions in total.
4. Pick codevector of each M codecell as the centroid for that codecell.

In Fig. 2.11, there is an example of SSQ, where x-axis was quantized first independently and then y-axis using the conditional probabilities.

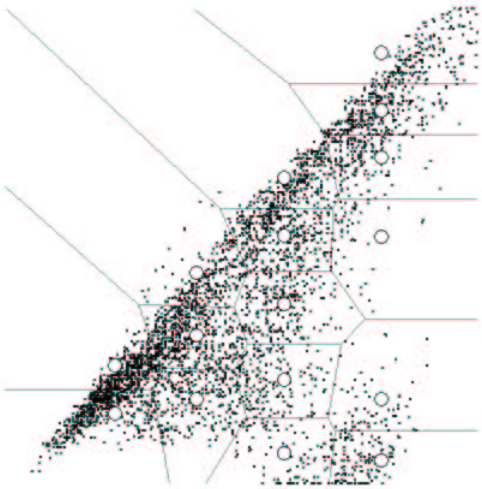


Figure 2.11: Example of sequential scalar quantizer on two dimensions (MSE = 157.84).

2.7.3 Histogram Based Algorithms

Popularity algorithm was invented by Tom Boyle and Andy Lippman at the Architechture Machine Group at MIT in the Summer of 1978 when Tom Boyle wrote the software called "Color Maker". Subsequently, Paul Heckbert rewrote that software with the name "IMAGE" and published the algorithm in [50]. Popularity algorithm is a simple and fast way to create color palette, it is also one of the first suggested algorithms in the field of color quantization.

The main idea is to count the occurrences of the colors in the image and then select M most frequent colors as the color palette. The Algorithm works in $O(N \log N)$ time, as picking M most frequent colors can be done by first sorting the pixel values and then scanning the sorted list. Same pixel values are consequtive entries in the sorted list, so scanning take's only $O(N)$ time. Basic assumption behind the popularity algorithm is that histogram peaks correspond to the cluster centroids. Unfortunately, this assumption does not hold in practice. Histogram might have fewer peaks than clusters or even more common, that one peak is much bigger than the others. In which many colors of same shade will be selected as codevectors. In Fig. 2.12, there is an example of this case.

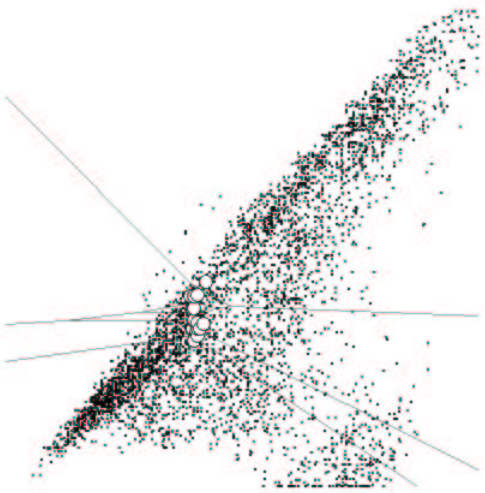


Figure 2.12: Example of popularity algorithm on two dimensions (MSE = 2750.93).

One possibility to alleviate the problem of picking similar colors as the codevectors is to select M most frequent colors that are separated by at least a predefined dis-

tance [52]. Then using the nearest neighbour condition, the rest of the colors are mapped to codevectors. In this way, the algorithm finds the clusters. Then centroid condition is used to generate final codevectors as the average colors of the corresponding clusters. Their idea is that the GLA method with one iteration and initial codebook with popularity algorithm. Generated solution could be finetuned by adding few a GLA iterations.

Some further enhancements to popularity algorithm have been proposed in [9], and it has been further modified to work in CIELUV color space [40]. One global histogram does not describe effectively local features of the image, Larabi *et al.* [65] proposed a method to calculate local windowed histograms and how to calculate the final palette from the collection local histograms.

2.7.4 Octree Quantization Algorithm

Octree quantization algorithm [42] is fast and produces comparable quality color palettes. Original paper was later reprinted in *Graphics Gems I* [43], which resulted in wide adoption of the algorithm in different implementations. Many commercial image processing packages are still using octree quantization.

Octree quantization is based on the *octree data structure* [82], which is a tree where each node contains pointers to eight children nodes. The colors are stored in the leaf nodes.

Octree quantization produces color palette in one pass. Also it does not use any prequantization of the input vectors. The idea is that image is processed pixel by pixel. The tree is traversed until a leaf node is reached, and if the leaf node is empty then pixel color is inserted into octree data structure; otherwise frequency count of that color is increased.

RGB colors have each 8 bits per component, so we can separate RGB color into 8 bitplanes of 3 three bits. For example, at bit position i R has a bit 1, G has bit 0 and B a bit 1. This leads to a 3 bit binary number $101_2 = 5_{10}$. It is clear that with three bits we can represent numbers from zero to seven. This number is used as a child node index. First, we take the most significant bitplane, use the index to traverse to one of the eight children nodes. Then we take the next bitplane and again use that as a index of a child node. Traversal continues until we take the last bitplane at the level eight, which is the leaf node.

When all pixels have been processed, then the process of reducing the leaf nodes start. The algorithm searches for a node, for which the sum of the frequencies is minimum. It calculates the average vector of all leaf nodes under this node. Removes all leaf nodes and stores the new average vector and the summed frequencies. The process continues until only M leaf nodes are left.

Reduction operator divides the color space to cubes of different sizes. The size of the cube depends on the level in which the octree reduction was performed. Higher up in the hiertarchy larger is the cube. As seen in previous example cube or square is not optimal partition shape, thus distortion from octree quantization is larger than competing methods [81].

2.7.5 Generalized Lloyd algorithm (GLA)

Generalized Lloyd algorithm (GLA) [70] is without doubt the most known codebook generation algorithm. It is known by many names, for example LBG from the initials of the original authors, and *C-means* algorithm [74] in statistical literature, where C refers to the number of clusters (it is also known by the name K-means). The name GLA comes from the fact that it is the generalization of the Lloyd's scalar quantization algorithm [72] to vector quantization.

GLA is derived from two necessary conditions for codebook optimality, namely the nearest neigbour and the centroid condition. GLA is an iterative algorithm, which starts from an initial solution. This solution can be generated, for example, by randomly picking M vectors from the dataset, or using some fast clustering algorithm to generate an approximate solution. The idea is that in each iteration GLA uses the nearest neighbour condition to produce the optimal partitioning of the training set given the codebook from the previous iteration. Then it uses the centroid condition to generate a new codebook.

GLA is a *descent* optimization algorithm, where distortion is not increased in successive iterations. Thus, the algorithm converges to a *locally optimal solution* but the convergence to the *globally optimal solution* is not guaranteed. Final quality of the codebook is dependent on the initial solution [58]. The practical solution to local optimality is to run GLA multiple times with many different random initial solutions and then pick the best codebook. In Fig. 2.13, on the right, we see a good solution with MSE of 101.64 and in the left we see result of the GLA with one random initial

solution with MSE of 114.80. Of course, if we are lucky we can get low MSE with just one random initial solution.

Algorithm 1 GLA

Generate initial codebook C

repeat

$S \leftarrow$ Generate optimal partitioning for a given C

$C \leftarrow$ Generate optimal codebook for a given S

Fill empty partitions

until Convergence

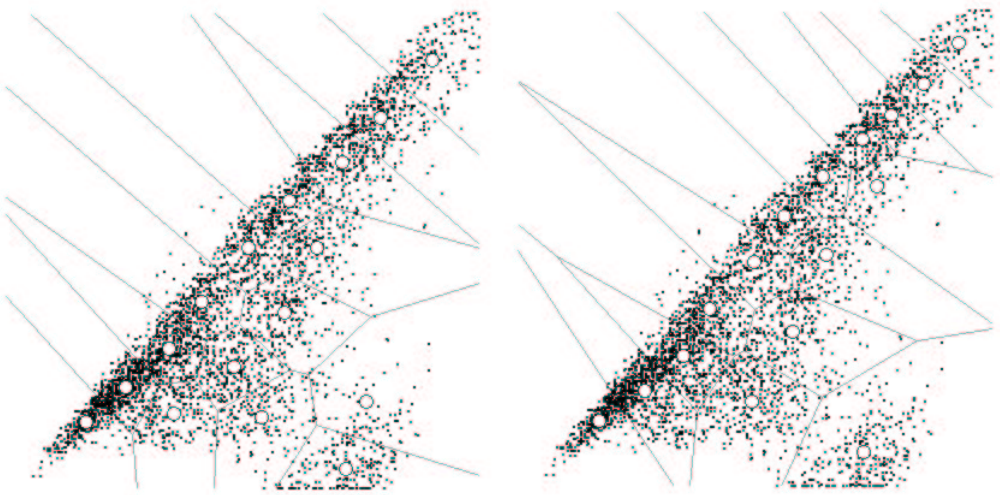


Figure 2.13: An example of GLA algorithm, left with one initial solution (MSE = 114.80) and right with 30 random initial solutions (MSE = 114.80).

As each GLA iteration is guaranteed not to decrease the quality of the solution, many codebook generation methods use GLA as a postprocessor that *fine-tunes* the generated codebook [39, 103, 77].

Related algorithm to GLA is ISODATA [5], which was used for color quantization in [19]. Iterative fine-tuning works exactly same way as in GLA, but if cluster is large enough it is split in two and if two clusters are close in enough they are merged to one. Large and close enough are user specified parameters. As codebook size is not set in the beginning ISODATA can be used in a situation, when number of color

clusters is not known in advance. Other option to solve number of clusters problem is to use F-ratio as a measure of *goodness of fit* of clustering model to that data set [63].

Chapter 3

Divisive Algorithms

"There are 2 kinds of people in this world,
and both are wrong!"
– anonymous

In this section, we present divisive clustering algorithms, which are also called in VQ literature *tree-structured codebook generation* (TSVQ) algorithms. Divisive method for vector quantization was first developed in [12]. Even though originally divisive methods were used in speech coding, it has been succesfully applied to color quantization. In general, divisive methods generate palette fast, with quality is comparable to the other palette generation methods [81].

Divisive methods are inspired by the *divide-and-conquer* [22] approach. Divide-and-conquer approach works for the problems, which can be recursively divided into smaller subproblems, which are similar to the original problem. Solving the subproblems and combining the solutions gives the solution to the original problem.

Divisive methods start with the codebook of one codevector representing the whole dataset. Partition consisting of the whole dataset is then divided into two partitions. Old centroid of split partition is discarded and two new centroids are formed as the centroids of the two new partitions. Process of division continues until the codebook size M is reached. Splitting can be done recursively as in [50] or iteratively, which will be the case in this thesis.

Divisive algorithms are conceptually simple, but details can be involving. Main question is where and what to split. First algorithm needs to decide, which partition to split. Usually the decision is a heuristic one, split the largest partition or the one

with the largest distortion. Local optimization strategy splits the partition, which decreases the distortion most [38].

For the decision on where to split we need to define two concepts namely *splitting axis* and *splitting plane*. Splitting plane is the plane that splits partition S_i in two, and splitting axis is orthogonal to it. Divisive algorithms work by first selecting splitting axis and then finding a suitable splitting plane. Equivalent of finding the splitting plane is finding the *cut-point* in the splitting axis, where splitting plane will be placed.

To sum-up, in each iteration every divisive algorithm makes following decisions:

1. Choose partition to split
2. Choose splitting axis
3. Choose cut-point

In computer graphics community, divisive methods are called *binary space partition* (BSP) methods [27], with corresponding data structure. Idea is similar to the divisive methods, but geometric primitives are stored in the tree data structure instead of vectors (points). For example stored primitives can be line segments. Typical BSP algorithm limits splitting hyperplane to be orthogonal to one of the coordinate axes, which is called in the BSP literature *axis-parallel cutting lines* [26].

Many different splitting strategies have been proposed in VQ literature [108, 12, 104, 38], and a few in papers that directly deal with color quantization [109, 81, 71, 55, 77]. In the following, is short exposition on a few different splitting strategies.

3.1 The Number of Splitting Planes

Let's consider the case, when splitting axis is fixed. Then we can investigate, the number of possibilites of splitting the dataset in two. First we project all vectors to the splitting axis. Remembering the restriction that the splitting plane is orthogonal to the splitting axis, we can take the splitting axis and move it to the position where there is one projected vector in the other side and all the rest in the other. Moving splitting plane one vector at the time to the other direction leads to the $N-1 = O(N)$ different possibilities for the placement of the splitting plane [38].

The problem of choosing the splitting plane is then moved to how to select splitting axis. In this section, we want to find out, for a purely theoretical interest, the number

of splitting planes, without restricting splitting axis. We formulate the question in following way: how many different ways can 3-dimensional set be split in two? If answer to the question is small enough, then enumerating all possible bi-partitions is possible. But we will see, that this is not the case.

We borrow the following result from the pattern recognition literature. First we need a definition of a term *general position* [27]. Vector set is defined to be in general position if there is no subset of $D+1$ of vectors that lie on an $(D-1)$-dimensional hyperplane. For example, two dimensional space having 3 vectors distributed on a line. We have to note that general position assumption might not hold for all datasets [27]. If this holds then the number of splitting planes in color quantization case is [24]:

$$C(N) = \begin{cases} 2\sum_{i=0}^{3} \binom{N-1}{i}, & N > 4 \\ 2^N, & N \leq 4 \end{cases} \tag{3.1}$$

where

$$\binom{N-1}{i} = \frac{(N-1)!}{(N-1-i)!i!}. \tag{3.2}$$

We can find the order of $C(N)$ by first forgetting the coefficient 2:

$$\begin{aligned} C(N) &= 1 + \frac{(N-1)!}{(N-2)!1!} + \frac{(N-1)!}{(N-3)!2!} + \frac{(N-1)!}{(N-4)!3!} \\ &= N + \frac{O(N^2)}{2} + \frac{O(N^3)}{6} = O(N^3). \end{aligned} \tag{3.3}$$

This result implies that enumerating all possible bi-partitions very fast grows to be unfeasible. For example, let's call N_1 size partition before split and N_2 size of new partition. Then the time complexity of enumerating all possibilities of two consecutive splits is $O(N_1^3 N_2^3)$. And if $N_1 \approx N_2$, then the time complexity would be $O(N^6)$.

3.2 Median Cut

Median cut was first proposed by Heckbert in his classic color quantization paper [50]. His idea is to split color space recursively into rectangular partitions. First, algortihm projects all vectors to the coordinate axes. Splitting axis is the coordinate axis i such that

$$\max_{i \in \{R,G,B\}} (\max(X_i) - \min(X_i)), \tag{3.4}$$

where X_i is the projected dimension. Median of the projected values is the cut-point of that partition. This leads to equal number of colors in both new partitions S_1 and S_2.

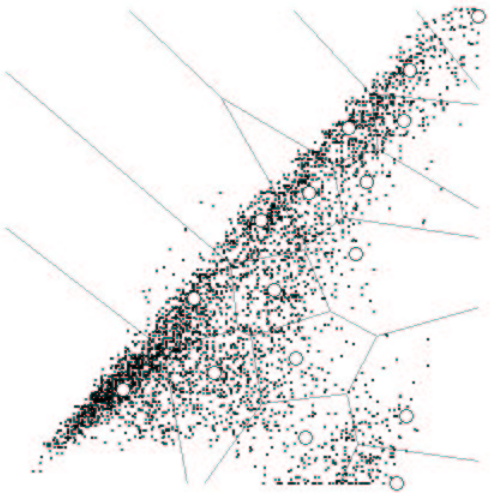


Figure 3.1: An example of Median Cut algorithm (MSE = 168.71).

Splitting process continues recursively until M number of partitions are created. Then partition centroids form the codebook. Heckbert proposed, that GLA could be used to fine-tune the solution generated by the splitting process. Original algorithm [50] is recursive, but iterative variant is easy to define. Here we present the iterative variant.

Algorithm 2 Median Cut

$S_1 \leftarrow$ smalles RGB cube that covers all training vectors
$M \leftarrow 1$
repeat
  $S_i \leftarrow$ select which maximizes $\max(X_i) - \min(X_i)$
  $X_i \leftarrow$ find axis from S_i which maximizes $\max(X_i) - \min(X_i)$
  $x_j \leftarrow$ find median X_i
  $S_{2i}, S_{2i+1} \leftarrow$ split S_i from x_j orthogonal to X_i
  Calculate codevectors $\boldsymbol{c}_{2i}$ and $\boldsymbol{c}_{2i+1}$ using Eq. (2.16)
  $M \leftarrow M + 1$
until $(M = |C|)$

Heckbert [50] uses *radix sort* to find the median, which requires $O(N)$ time, when number of digits in the numbers to be sorted is constant [22]. However *median selection* without sorting can be perfomed in $O(N)$ [62]. Total time complexity of the median cut is then $O(N \log M)$.

Many variations exists of median cut algorithm, for example *center-cut* [55] algorithm splits from the center of the partition instead of median. Splitting in the mean avoids the extra computational cost of finding the median.

3.3 Variance-based Algorithm

Assumption behind the median cut is that all partitions should have approximately equal number of vectors. In practice, each division splits partition exactly in half if the number of vectors in that partition is even. However there is no real justification for this assumption. For example, when partition has two regions, where one is a dense concentration of vectors and the other is less dense. Then it is not sensible to split dense region only for the reason that new partitions should have equal amount of vectors.

Problems in Median Cut split strategy were recognized by Wan *et al.* [97]. Their method [96, 97] is iterative and it is based on optimization stategy, where in each iteration split is perfomed on such a way that the decrease in distortion is maximized. Reitan [81] calls this algortihm an *variance minimization* technique. And variance of the partition S_i is defined:

$$\sigma_i^2 = \frac{1}{\sum_{\boldsymbol{x}_j \in S_i} p_j} \sum_{\boldsymbol{x}_j \in S_i} p_j ||\boldsymbol{x}_j - \boldsymbol{c}_i||^2. \tag{3.5}$$

One dimensional case this is true, but in 3-dimensional case σ_i^2 represents the average distortion of the partition S_i. Variance minimization algorithm selects partition, which has the highest average distortion to be split. Later term $\frac{1}{\sum_{\boldsymbol{x}_j \in S_i} p_j}$ is dropped, which leads to σ^2 being the total distortion of the partition (TSE).

In variance minimization, split axis is one of the coordinate axis and the cut-point is the point in projected coordinate axis, which will yield largest reduction in distortion. Split axis is selected to be the coordinate axis, which used as a split axis will yield largest reduction in distortion.

Colors are projected into each coordinate axis of partition S_i, these colors then form 1-dimensional distribution. Then the optimal cut point for that partition is [97]:

$$t_{\text{opt}} = \arg \max_{\frac{\mu+\text{lower}}{2} \leq t \leq \frac{\mu+\text{upper}}{2}} \frac{w_1}{w_2}(\mu - \mu_1(t))^2, \tag{3.6}$$

where μ is mean and $\mu_1(t)$ is the mean of the new projected distribution for the first half if the cut point is at t. w_1 and w_2 are the the frequencies of the tentative new partitions S_{i+1} and S_{i+2}. The variables *upper* and *lower* are max and min of the projected distribution. Pseudo code for variance-based split is in algorithm 3.

Algorithm 3 Variance-based split

$M \leftarrow 1$
repeat
  Find partition S_i, with largest distortion Eq. (3.5)
  Project all colors to its coordinate axes
  Calculate cut-points t_R, t_G, t_G with (3.6)
  Split S_i from cut point t_i
  $M \leftarrow M + 1$
until $M = |C|$

In *greedy tree growing* (GTG) [71] method, partition to be split is selected locally optimally. GTG method precalculates for each partition the amount of distortion will be decreased if that partition is split. Other partitions around the partition in question do not affect the decrease in distortion. That is the reason, why it can be saved. Then in each iteration algorithm selects the partition, which split results largest decrease in distortion. Split axis and cut-point selection are done the same way as in the variance minimization scheme.

3.4 Orchard's Binary Splitting Algorithm

Orchard and Bouman proposed in [77] a divisive palette generation algoritihm, which uses a local optimization strategy in selecting the splitting axis. Previous splitting algorithms were limited to select splitting axis from one of the RGB coordinates.

Partition to be split is selected to be the one, which has largest distortion. This can be approximated, by the largest *eigenvalue* of the *covariance matrix* (see details in Appendix A) of the vectors in that partition [77]. In Orchard's method, partition with the largest eigenvalue is chosen to be split.

Idea in Orchards method is to find the splitting axis in such a way that minimizes the distortion. Intuitively such a axis is the direction or largest variance of that partition. This axis can be found with *principal component analysis* and it is called the principal axis or the eigenvector corresponding to the largest eigenvalue. More details on the principal component analysis can be found from Appendix A. Cut-point is selected to be the projection of the centroid to the principal axis.

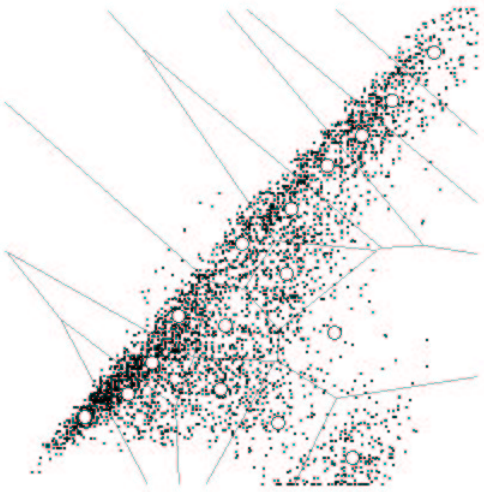


Figure 3.2: An example of Orchard's binary splitting algorithm (MSE = 115.32).

Problem with their method is that it assumes that best cut-point point is the projection of mean vector. The best case can happen, when two clusters are perfectly divided by that plane. It is easy to construct a counterexample, where that does not hold. For example see. the Fig. 3.3, where splitting plane is orthogonal to principal axis and is going through the centroid. Centroid as cut-point is clearly sub-optimal in this case.

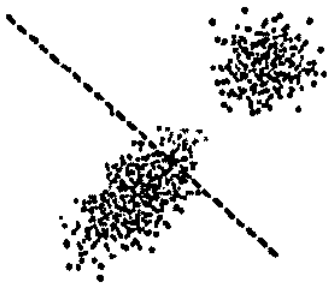


Figure 3.3: Example of when the centroid is not the optimal cut-point.

Algorithm 4 Orchard's method

$M \leftarrow 1$
repeat
  Find partition S_i, with largest distortion
  Calculate principal component of partition S_i
  Cut point t is the projection of $\boldsymbol{c}_i$ to principal component
  Split S_i from cut-point t
  $M \leftarrow M + 1$
until $M = |C|$

3.5 Radius Weighted Mean Cut

Radius weighted mean cut (RWM Cut) [108] is the vector quantization split strategy orginally used for *Block Truncation Coding* (BTC) image compression [106]. Later the same method was applied to color quantization [109].

RWM Cut selects the partition with largest distortion to be split, as was done in previous method. Also splitting axis is the principal axis of the partition. The cut-point is defined as the projection of the *radius weighted mean* (RWM) vector to the principal axis. RWM vector is defined as [108, 109]:

$$\boldsymbol{c}_i' = \frac{1}{\sum_{j=1}^{|S_i|} ||\boldsymbol{x}_j - \boldsymbol{c}_i||} \sum_{j=1}^{|S_i|} \boldsymbol{x}_j ||\boldsymbol{x}_j - \boldsymbol{c}_i||. \tag{3.7}$$

Other possibility for a splitting axis is presented in [109], as the line going through the $\boldsymbol{c}_i$ and the $\boldsymbol{c}_i'$. As calculating the $\boldsymbol{c}_i'$ is $O(N)$ operation, the splitting axis selection

strategy would be very fast. It is interesting to note that in few test images the line going through $\boldsymbol{c}_i$ and $\boldsymbol{c}_i'$ is very close to the principal axis [81]. But in general no proof of local optimality is given for this splitting axis selection strategy nor cut point selection strategy.

Other approach was recently described by Shyi-Chyi *et al.* [85], where authors defined splitting axis to be a line going from $\boldsymbol{c}_i$ to the vector which is

$$\arg\max_{\boldsymbol{x}\in S_i} ||\boldsymbol{x} - \boldsymbol{c}_i||. \tag{3.8}$$

All vectors in the partition are projected to this line, then the cut point is found by using two level scalar quantizer on projected values. Vectors, which projected value falls in the class 0 form a partition and the vectors which projected value is in class 1 form the other partition. Cut point is the decision boundary between the classes 0 and 1. All partitions are split whose variance of distances from vectors to the partition centroid is above the user defined threshold.

3.6 Locally Optimal Bi-partitioning

In *locally optimal bi-partitioning* [104, 38], partition to be split is the one that results in the largest reduction in distortion. Strategy is the same as in greedy tree growing. Splitting axis is the principal axis. And the cut point is found optimally by considering all cut point position t in the principal axis such that [104]:

$$t_{opt} = \arg\min_{\xi}\{D(S_1(t)) + D(S_2(t))\}, \tag{3.9}$$

where $D(S_1(t))$ is the distortion of the partion S_1 formed from the cut point t. It has been shown, that t_{opt} can be found in $O(N)$ time if projected distances were sorted first [104].

After optimal cut point is found, tentative partitions S_1 and S_2 are formed. Then *local repartitioning* with GLA is used to fine tune the partition boundary between the partitions S_1 and S_2.

Algorithm 5 Locally optimal bipartitioning

$M \leftarrow 1$

repeat

Find partition S_i, which split decreases distortion most

Calculate principal component of partition S_i

Find optimal cut point t

$\{S_{2i}, S_{2i+1}\} \leftarrow$ partition S_i using the cut point

Use GLA to refine the partioning $\{S_{2i}, S_{2i+1}\}$

$M \leftarrow M + 1$

until $M = |C|$

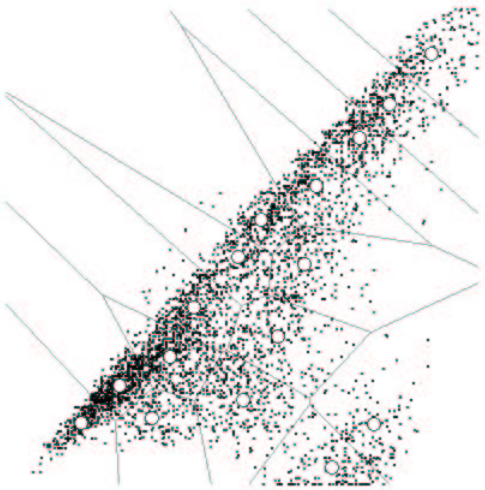


Figure 3.4: An example of locally optimal bipartitioning (MSE = 106.26).

3.7 Optimal Principal Quantizer

All previous algorithms in this chapter used the bi-partitioning strategy, one partition is always split in just two partitions. In this section, we will lift this restriction, resulting many splits at the same time. It is well known that the colors in the color space are spread along the *luminance* axis [47]. Consequently luminance axis is also the principal axis of the dataset [101].

In [101], Wu proposed an *optimal principal quantizer*, which first projects all vectors

to the principal axis and then sorts the vectors according to their projections. Now if we consider the projection values we have a list of scalar values representing the whole color space. Optimal principal quantizer uses optimal scalar quantizer [100] on the projection values to partition the set of projections into the k intervals, where $k < M$. Partitioning of the projected values induces a k partitioning of the color space. Distortion function used in the scalar quantization of projection values is the distortion of quantizing the vectors in that partition. In that way optimal principal quantizer can be considered as a hybrid method of two existing methods: scalar quantization and vector quantization [101]

The value of k is determined dynamically as the bottom-up dynamic programmig process continues [101]. If after optimal principal quantizer is perfomed $k < M$, then $M - k$ partitions still need to be formed. Algorithm then uses locally optimal bipartitioning scheme to split those partitions created in the previous stage.

Chapter 4

Agglomerative Algorithms

"Hypothesis: All odd numbers are prime
Proof: If a proof exists, then the hypothesis must be true
The proof exists; you're reading it now.
From 1 and 2 follows that all odd numbers are prime"
– anonymous

Agglomerative clustering [8] algorithm generates clusters by a sequence of merge operations. Agglomeration process starts by initializing each data vector as its own cluster. Two clusters are merged at each step and the process is repeated until the desired number of clusters has been obtained. If we are considering the general clustering problem, then many different merge rules exist. Different merge rules give different clustering solutions, and thus those stragegies assume different cluster shapes [89]. The *single linkage* (SL) [86] method determines the cluster pair to be merged based on the two closest vectors. The *complete linkage* (CL) [86] method determines the cluster to be merged based on the two furthest vectors. The *unweighted pair-group method using averages* (UPGMA) [89] determines the unweighted average distance between all intercluster vector pairs. The *weighted pair group method average* (WPGMA) [89] considers the weighted average distance between all intercluster vector pairs, weight is selected in such a way that vectors in the larger cluster receive lower weight and two clusters of size one receive both weight one. *Ward's* method [99] selects the cluster pair to be merged in such a way that increases the distortion least. In VQ context, this is known as the *pairwise nearest neighbour* (PNN) method [33]. In this section, we consider only the PNN merge method.

4.1 PNN Merge Distance Function

The overall goal in the vector quantization is to minize distortion cost function, for example MSE or TSE distortion. The same also applies to the PNN algorithm so that in each iteration, the function would be minimized. We know that distortion is monotonicly decreasing as a function of number of partitions (centroids) [41]. This means that in each PNN iteration overall distortion is increased. For that reason, we have to find for all pairs or partitions S_i and S_j, how much their merge would increase overall distortion.

Ward proved in [99] that given TSE as a cost function, optimal choice S_a and S_b for merging is when following is minimal:

$$D(S_{ab}) = D(S_a) + D(S_b) + \frac{n_a n_b}{n_a + n_b}||\boldsymbol{c}_a - \boldsymbol{c}_b||^2, \tag{4.1}$$

where n_a and n_b are frequencies of the respective partitions, and $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ are their centroid vectors. Thus, increase of distortion, when merging two partitions can be determined only by the frequencies of the merged clusters and their centroids.

The optimality of a single merge operation does not necessarily lead to a optimality of the sequence of the optimal merge operations. Thus PNN method follows a *local optimization strategy.*

Now we can define PNN *distance* $d_{\mathrm{PNN}}(S_a, S_b)$ as the increase of the distortion, when clusters S_a and S_b are merged.

$$d_{\mathrm{PNN}}(S_a, S_b) = \frac{n_a n_b}{n_a + n_b}||\boldsymbol{c}_a - \boldsymbol{c}_b||^2, \tag{4.2}$$

As noted by the symbol $d_{\mathrm{PNN}}(,)$ PNN merge cost can be thought to be a distance function between two partitions (or clusters in general clustering problem). Next we analyze some properties of this distance function. Sometimes it is usefull if distance function used is a *metric.* If it is metric, then it has to satisfy metric space axioms:

Axiom 1. Symmetry: $d(a, b) = d(b, a)$.

Axiom 2. Non-negative: $d(a, b)) \geq 0$.

Axiom 3. Non-zero: $d(a, b) = 0 \quad \Rightarrow \quad a = b$.

Axiom 4. Triangular inequality: $d(a, c) \leq d(a, b) + d(b, c)$.

Lemma 1. $d_{\mathrm{PNN}}(.,.)$ *does not follow triangle inequality*

Proof. We prove by counter example. Let's take $\boldsymbol{a} = (1,0)^t$, $\boldsymbol{b} = (0,1)^t$ and $\boldsymbol{c} = (0,0)^t$ and corresponding cluster frequencies are $n_a = 3$, $n_b = 1$ and $n_c = 3$. Then $d_{\mathrm{PNN}}(\boldsymbol{a},\boldsymbol{b}) + d_{\mathrm{PNN}}(\boldsymbol{b},\boldsymbol{c}) = 1.9313$ and $d_{\mathrm{PNN}}(\boldsymbol{a},\boldsymbol{c}) = 2$, which cannot be true according to triangle inequality, and, thus distance $d_{\mathrm{PNN}}(.,.)$ is not a metric. □

What does this mean in practice? It is true that triangle inequality is not necessary for the basic operation of PNN algortihm, but if triangle inequality would be satisfied then we could use *metric space indexing* [16] techniques to find the pairwise closest partitions faster. Normally it is necessary to consider all pairwise distances to find closest pair.

4.1.1 Merging All Vectors to the Same Partition

Now we consider the case, when every vector in the dataset is iteratively merged to the same cluster. We call this cluster S_1. Merging all vectors one by one to S_1 means that the merge cost of vector $\boldsymbol{v}_i$ to cluster S_1 is:

$$\frac{n_1}{n_1+1}||\boldsymbol{c}_1 - \boldsymbol{c}_i||^2, \tag{4.3}$$

where n_1 is the number of vectors already merged to the cluster S_1. Now if we take a limit of $n_1 \to \infty$ of the coefficient $c = \frac{n_1}{n_1+1}$.

$$\lim_{n_1\to\infty} \frac{n_1}{n_1+1} = \lim_{n_1\to\infty} \frac{n_1}{n_1(1+\frac{1}{n_1})} = 1. \tag{4.4}$$

we get unity. This means that when n_1 is "big", we can forget the coefficient c from d_{PNN}, reducing it to the squared distortion. For example with 200 iterations we would get $c = 0.99502$. Figure 4.1 shows that the coefficient c converges very fast closer and closer to the unity.

This leads to the hypothesis that with large partition sizes coefficient could be forgotten. Also using l_2 distance instead of squared error in that case, fast indexing techniques could be used to find the closest pair of all partitions.

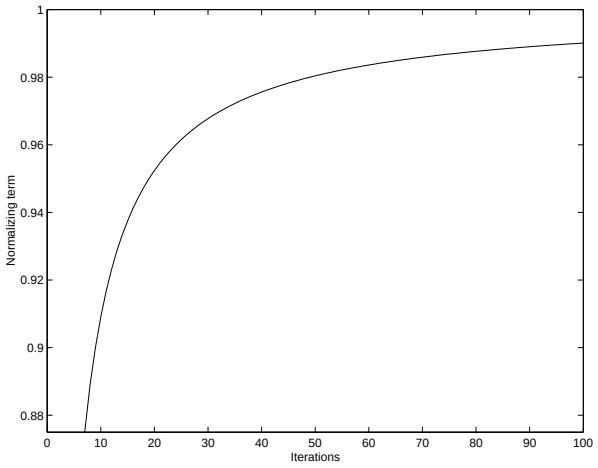


Figure 4.1: Coefficient c as a function of iteration.

4.1.2 Monotony Property

Interesting monotony property describing the growth of PNN distances was proven in [59]:

Lemma 2. *Consider the clusters S_a, S_b and S_c, with centroids $\boldsymbol{c}_a$, $\boldsymbol{c}_b$ and $\boldsymbol{c}_c$ and cluster frequencies n_a, n_b and n_c. Assume that $d_{\mathrm{PNN}}(S_a, S_b) \leq d_{\mathrm{PNN}}(S_a, S_c), \leq d_{\mathrm{PNN}}(S_b, S_c)$ and $n_a, n_b, n_c \geq 1$. Then it holds that $d_{\mathrm{PNN}}(S_a, S_c) \leq d_{\mathrm{PNN}}(S_{ab}, S_c)$.*

In other words, distance from cluster S_i to another cluster S_j can never be smaller after S_j has been merged to some other cluster. This means, that PNN distances are monotonously non-decreasing when PNN iterations are progressing. It was noted in [14], that above mentioned monotony criterion is also true for any type of symmetric and totally ordered distance.

4.2 Overview of Agglomerative Methods

The PNN algorithm [33] is described in Algorithm 6. In each iteration, it merges two partition that minimizes distortion. Number of iteations is $N - M$, and as M (codebook size) generally is much smaller than N ($M \ll N$), the number of iterations is $O(N)$. To find such two clusters, whose merge minimizes distortion, we only need to find two partitions S_a and S_b, whose $d_{\mathrm{PNN}}()$ distance is minimum. See section 4.1

for more details on the PNN distance function. "Merge(S_a, S_b)" -operation on the other hand calculates new centroid for a new cluster [33]:

$$\boldsymbol{c}_{a+b} = \frac{n_a \boldsymbol{c}_a + n_a \boldsymbol{c}_a}{n_a n_b}, \tag{4.5}$$

where n_a and n_b are the corresponding cluster frequencies. Then cluster frequency of the new cluster is simply $n_{a+b} = n_a + n_b$.

Algorithm 6 PNN method

Mark every data vector to be codevector ($M \leftarrow N$)
repeat
  $(S_a, S_b) = \text{FindNearestClusters}(S)$
  $S_{a+b} = \text{Merge}(S_a, S_b)$
  $M \leftarrow M - 1$
until $M = |C|$

The number of all pairwise distances is $\binom{N}{2}$, as binomial coefficient describes how many ways we can select two from N. Which leads to

$$\binom{N}{2} = \frac{N!}{(N-2)!2!} = \frac{1}{2}(N-1)N = O(N^2), \tag{4.6}$$

where we see, that with trivial algorithm by enumerating all pairwise distances has the time complexity of $O(N^2)$. Which yields to the $O(N^3)$ time complexity of the whole PNN algorithm When we consider more carefully the operations inside the PNN iterations, we find that all are distance computations [83]. For arbitrary dimension D this will lead to $O(DN^3)$ time complexity. But fortunately in color quantization we quantize only 3-dimensional vectors. This method we call *exact* PNN.

Several speedups have been proposed in the literature to exact PNN. One possibility to reduce the distance calculations is to use a distance matrix A, where an element $a_{ij} = d_{\text{PNN}}(S_i, S_j)$. Matrix is upper triangular as d_{PNN} is symmetric. When partitions S_i and S_j are merged, only row i and column j needs to be recalculated. Which leads to $O(N)$ update operation. Authors in [92] propose to use exact PNN variant with distance matrix for color quantization. Kurita [64] proposed to use distance matrix for updates and heap structure for finding the minimum pairwise distance in $O(\log N)$ time. Time complexity for Kurita's method is $O(N^2 \log N)$, but unfortunately from keeping all pairwise distances in the heap and in the distance matrix the space complexity becomes $O(N^2)$.

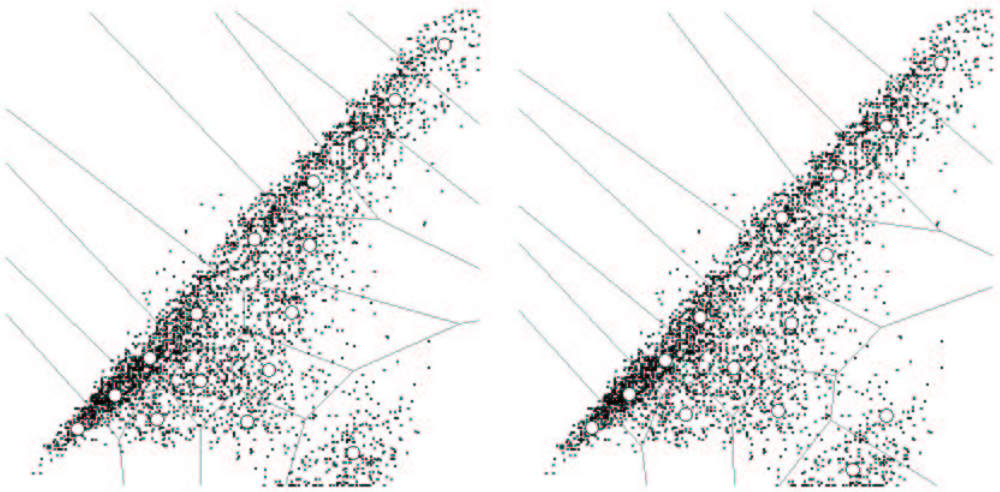


Figure 4.2: An example of PNN algorithm on the right(MSE = 108.64) and on the left PNN solution fine tuned with GLA (MSE = 101.94).

A linear in space complexity, but still order of magnitude faster than exact PNN algorithm has been introduced, called *fast exact PNN* [39]. The idea is to calculate nearest neighbour pointer $nn()$ to all partitions:

$$nn(S_a) = \arg \min_{i \in [i,N], i \neq a} d_{\text{PNN}}(S_a, S_i). \tag{4.7}$$

To find minimum distance in $d_{\text{PNN}}()$ sense only $O(N)$ steps are needed in each iteration. After each merge operation, nearest neighbour pointers have been updated to point to a new possibly closer partitions. This method is faster than previous ones but it is still lower bounded by $\Omega(\tau N^2)$, where τ denotes the number of partitions whose nearest neighbour pointer must be updated.

Lemma 3. *if* $d_{\text{PNN}}(S_a, nn(S_a))$ *is minimum over all pairwise distances and* $nn(S_a) = S_b$*, then* $nn(S_b) = S_a$

Proof. We prove by contradiction. Let's suppose that $nn(S_b) \neq S_a$, then $d_{\text{PNN}}(S_b, nn(S_b)) < d_{\text{PNN}}(S_b, S_a)$. But this cannot hold as $d_{\text{PNN}}()$ is symmetric and thus $d_{\text{PNN}}(S_a, S_b) = d_{\text{PNN}}(S_b, S_a)$. □

This means that in each iteration only cases when $nn(S_a) = S_b$ and $nn(S_b) = S_a$ need to be checked. Separate list holding pointers to those partitions which have this

property. Then find minimum operation needs only to go through this list.

Other possibility is to use *dynamic closest pair* data structure, where update cost is $O(N \log^2 N)$ [31]. Total time complexity for clustering is then $O(N^2 \log^2 N)$ and space complexity is $O(N)$. Speedup of around 35% can also be gained compares to the fast exact PNN by deferring the distance computations until necessary [59, 14]. Also different practical methods to speedup PNN have been proposed the [95]. All of these methods still suffer the problem of quadratic time complexity.

To overcome the quadratic search cost Equitz proposed an approximative variant called *fast* PNN [33]. Fast PNN uses KD-tree to search partitions to merge and it also merges many partitions at the same time. Unfortunately quality of the produced codebook is not comparable to the exact PNN [39]. Also essentially the same algorithm as the fast exact PNN has been proposed for color quantization [3], in addition of using activity weighted metric instead of the MSE. By assuming color quantization as a special case of PNN algorithm Xiang *et al.* [107] consider a reduced set of candidates for pairing for each color instead of all pairwise partition distances. Idea is to search possible pair to merge in the small RGB boxes, where candidate pairs for each color should be found. This obviously produces suboptimal results compared to exact PNN but unfortunately the authors did not compare their approximative algorithm to exact PNN.

4.3 Agglomerative Clustering Using kNN Graph

Exact, but fast methods, still use *linear* time per PNN iteration, natural question then is that could we do better? In [30, 29], authors propose a method to do reduced search by graph. First they sort RGB triplets by their luminance and then calculating *k Nearest Neighbour Graph* (kNNG) [32], from the sorted luminance values. Independently of them, in [37] use of directed graph to solve quadratic time complexity was proposed. Main topic in the rest of this chapter is to delve more thoroughly in the details of the kNNG based PNN.

The proposed method is basically the same as the PNN method except that we utilize graph structure in the search of nearest neighbor. In the PNN, the search of nearest neighbor cluster is repeated many times, and every search requires $O(N)$ distance calculations. The graph is utilized so that the search is limited only to the clusters that are directly connected by the graph structure. If the number of edges is small,

significant reduction in computation can be obtained.

Algorithm 7 PNN on kNN Graph

Mark every data vector to be node in the directed nearest neighbour graph ($M \leftarrow N$)
Find $kNN(\boldsymbol{x}_i) \quad \forall \boldsymbol{x}_i \in X$
repeat
  $(S_a, S_b) = \text{FindNearestClustersInGraph}(S)$
  $S_{a+b} = \text{Merge}(S_a, S_b)$
  Find the k nearest neighbours of S_{a+b}
  Update clusters that have S_a and S_b as neighbours
  $M \leftarrow M - 1$
until $M = |C|$

Algorithm 7 shows the basic outline of the PNN algorithm with kNNG, where kNNG is first calculated and then used. We use the shorthand Graph PNN for this algorithm from now on. We clearly see that algorithm outputs the same codebook as the PNN if $k = N - 1$. Then graph is complete, and that matches the condition of the PNN algorithm. We can store kN distances in the heap structure as seen in Kurita's method [64], then the "FindNearestClustersInGraph()" -operation is $O(\log kN) = O(\log k + \log N) = O(\log N)$. Merge operation unchanged from the original PNN algorithm. In find the k nearest neighbours operation, k neighbours are picked from the neighbourhood lists of both S_a and S_b. When neighbours have been picked, distances to them are recomputed. For that reason heap structure has to be updated, which has a time complexity of $O(\log N)$.

Update clusters that have S_a and S_b as neighbours operation is the critical part of the algorithm. To find the clusters,which consider S_{a+b} as a neighbour, in constant time, algorithm maintains a incoming edges list. Size of the list is denoted by the symbol ϑ. Then the time complexity of that operation is $O(\vartheta \log N)$, where $\log N$ is again the heap update cost. Total cost of algorithm is then $O(\vartheta N \log N)$.

Now interesing problem is, can we give an estimate of ϑ. It would be advantageous if τ would be small constant, then the time complexity would be $O(N \log N)$. Unfortunately no such guarantee exists in $d_{\text{PNN}}()$ space [39]. Our definition of the nearest neighbour graph says that, at the graph creation phase, distance between vectors is Euclidean. Also assumption is that in the original dataset frequencies were one for each unique vector. It has been proven in [75] that ϑ is bounded by $\tau_D k$, where τ_D is

the *kissing number* of the dimension D [21]. Kissing number, which is defined as the maximum number of unit hyper spheres that are touching another unit hyper sphere without any interstections. Kissing number problem for general D is still unsolved problem, but fortunately for the case $D = 3$ has been solved to be $\tau_3 = 12$ [21].

4.4 k-Nearest Neighbour Graph

We define *k-nearest neighbour graph* (kNNG) as a weighted directed graph, in which every node represents a single vector, and the edges correspond to pointers to neighbour vectors. Every node has exactly k edges to the k nearest vectors according to a given distance function. The distance used herein is normal euclidean distance. First we need one auxiliary definition of neighbourhood set [10] before we jump into the formal definition of the kNNG. Let's first impose a well defined order in the set X. For all vectors in the set $\boldsymbol{q}_i \in X \setminus \{\boldsymbol{x}\}$, there exists following ordering:

$$d(\boldsymbol{x}, \boldsymbol{q}_1) \leq d(\boldsymbol{x}, \boldsymbol{q}_2) \leq \ldots \leq d(\boldsymbol{x}, \boldsymbol{q}_{N-1}). \tag{4.8}$$

This is trivially true, as distance $d(\cdot, \cdot)$ output's a real number and the set of real numbers is a well ordered. Now using the imposed ordering, we can define set $kNN(\boldsymbol{x})$.

Definition 1. *Set is $kNN(\boldsymbol{x})$, when let $k \leq N$ and $\boldsymbol{q}_i$ be as in Eq. 4.8, then:*

$$kNN(\boldsymbol{x}) = \{\boldsymbol{q}_1, \boldsymbol{q}_2 \ldots . \boldsymbol{q}_k\} \tag{4.9}$$

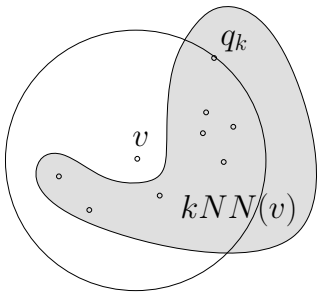


Figure 4.3: An example of the $kNN(\boldsymbol{v})$.

In Fig. 4.3, we see the geometrical interpretation of the $kNN(\boldsymbol{x})$, where $\boldsymbol{v}$ centered and $r = ||\boldsymbol{x} - \boldsymbol{q}_k||$ radius sphere is formed. Set $kNN(\boldsymbol{v})$ contains vectors in that sphere minus $\boldsymbol{x}$. Next we give formal definition of kNN graph. Let's define the set of edges from vector $\boldsymbol{x}$ as $e(\boldsymbol{x}) = \{(\boldsymbol{x}, \boldsymbol{q}_i) | \boldsymbol{q}_i \in kNN(\boldsymbol{x})\}$.

Definition 2. *kNNG is a directed graph* $G = (V, E)$*, where* $V = \{\boldsymbol{x}_i \in X\}$ *and* $E = \{e(\boldsymbol{x}_i) \in X\}$.

We see from definition 2 that as every vertex in a graph has exactly k outgoing edges, the number of edges is then $|E| = k|V|$. Note however, that using the kNNG with Graph PNN will reduce the number of edges in the graph. In each iteration, one vertex is taken out of the graph, then in the case when vertex $\boldsymbol{x}$ has an edge to the both vertexes to be merged one edge will be removed from $\boldsymbol{x}$'s edge list.

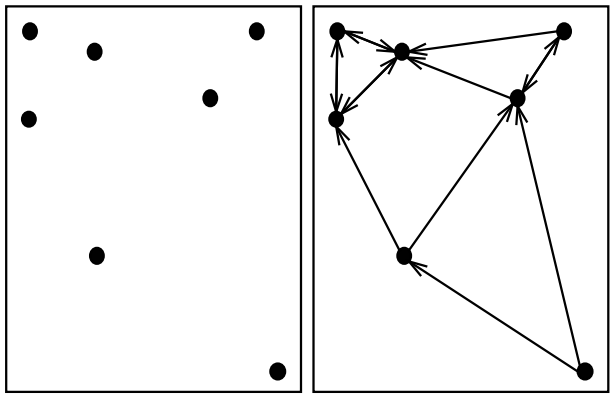


Figure 4.4: Sample $2d$ dataset (left), and the corresponding kNNG with $k = 2$ (right).

Fig. 4.4 give an example for kNNG for 2-dimensional dataset with k set to 2. Eppstein *et al.* [32] defined nearest neighbour graph, which is equivalent to our definition of kNNG, when $k = 1$. They also mentioned kNNG and concluded that graph theoretical and combinatorial properties of the kNNG are fairly unknown. Graph created with this definition has applications beyond the usage as auxiliary data structure in our Graph PNN algorithm. Some example are the use in *nonlinear dimensionality reduction* [88], where idea is to approximate the true *manifold* where high dimensional vectors reside. Previously we have also used the kNNG to detect *outliers* from vector set [48].

Other definitions of kNNG have been also proposed in literature. All of the graphs which deal with the notion of *neighbourhood* vector set are called *proximity graphs* [54], where vertices are vectors in the training set (or data set more generally) and edges between vertices describe spatial relationship of those vertices.

Mutual k Nearest Neighbour Graph (MkNNG) by Brito *et al.* [10] is defined formally as follows.

Definition 3. *MkNNG is an undirected graph* $G = (V, E)$*, where* $V = \{\boldsymbol{x}_i \in X\}$ *and* $E = \{(\boldsymbol{x}, \boldsymbol{q}), \boldsymbol{q}, \boldsymbol{x} \in X | \boldsymbol{x} \in kNN(\boldsymbol{q}) \wedge \boldsymbol{q} \in kNN(\boldsymbol{x})\}$.

Let's find the inclusion relation between the MkNNG and the kNNG.

Lemma 4. *MkNNG* $\subseteq$ *kNNG*

Proof. We prove by contradiction. By definition vertexes of both graphs are the same. Let's MkNNG $= (V, E')$ and kNNG $= (V, E)$. Then let's suppose, that there exists $e = (\boldsymbol{x}, \boldsymbol{q}) \in E'$ and $e \notin E$. Then by definition of MkNNG, $\boldsymbol{q} \in kNN(\boldsymbol{x})$, but also by definition of kNNG, $(\boldsymbol{x}, \boldsymbol{q}) \in E$, which is a contradiction. Also the case when $\boldsymbol{x} \in kNN(\boldsymbol{q})$ is proved in the same way. This completes our proof. □

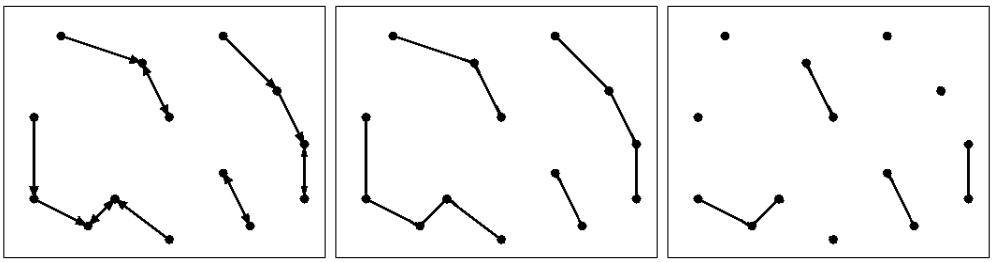


Figure 4.5: Example kNNG (left), the corresponding UkNNG (center) and the MkNNG (right) [2].

The case when $k = 1$ the MkNNG = kNNG can happen only when all vectors are each others nearest neighbours.

A different definition of *undirected k Nearest Neighbour Graph*[1] (UkNNG), it is formally defined as [75].

Definition 4. *UkNNG is an undirected graph* $G = (V, E)$*, where* $V = \{\boldsymbol{x}_i \in X\}$ *and* $E = \{(\boldsymbol{x}, \boldsymbol{q}), \boldsymbol{q}, \boldsymbol{x} \in X | \boldsymbol{x} \in kNN(\boldsymbol{q}) \vee \boldsymbol{q} \in kNN(\boldsymbol{x})\}$.

In other words, this means that vertices $\boldsymbol{v}$ and $\boldsymbol{q}$ have an *undirected* edge between each other if either of them consider the other as it's nearest neighbour. We use directed kNNG, but in principle UkNNG could also be used in agglomerative clustering. With kNNG we have the explicit knowledge which vertices $\boldsymbol{v}$ considers to be it's nearest neighbours.

[1] Authors use term k Nearest Neighbour Graph, but we coined a different term to distinguis it from kNNG.

Lemma 5. *If all edges in kNNG are changed to undirected edges and duplicate edges are removed, then $kNNG = UkNNG$.*

Proof. We prove this lemma by contradiction. Let and $G = (V, E)$ be a kNNG and $G' = (V, E')$ be a UkNNG. There are two cases, when lemma would not hold: i) $e \in E$ and $e \notin E'$ or ii) $e' \in E'$ and $e' \notin E$. Case i): Lets suppose that $e = (\boldsymbol{v}, \boldsymbol{q}) \in E$ and $e \notin E'$. Then $\boldsymbol{q} \in kNN(\boldsymbol{v})$, but by the definition of UkNNG $e \in E'$, which is a contradiction. Case ii) is satisfied symmetrically. □

Other possible proximity graphs are *Gabriel Graph* (GG), *Relative Neighbourhood Graph* (RNG), *Delaunay Triangulation* (DT) and *Minimum Spanning Tree* (MST). For detailed information of all of those, see review paper [54].

4.5 Neighbourhood Size

Graph PNN algorithm described in this section contains only one parameter, the neighbourhood size k. The question what is the correct value of k, has a practical importance as memory usage has *linear dependence* to k. If we want to be sure that the distortion of the produced codebook is exactly same as the codebook produced by the PNN algorithm, then the graph has to be complete. Graph is complete when $k = N - 1$. Space complexity of the neighbourhood graph is $O(kN)$, for that reason complete graph is not practical. We have to keep in mind that the exact PNN is also heuristic algorithm, which has only guarantees the optimality of the single merge operation. The sequence of the merge operations will not necessarily produce an optimal solution. On the other hand choice of $k = N - 1$ will lead to exactly same distortion as the original PNN method, as then the graph will be complete. Restricting neighbourhood graph to smaller than $N - 1$, will also cause some PNN merge sequences not to exist, then algorithm will select some other sequence. The non-optimality of the PNN method then states that when $k < N - 1$ algorithm can produce smaller distortion than the exact PNN. To summarize, $k = N - 1$ is not necessarily optimal in the distortion sense. However we are interested in the optimal choice of k in the time/distortion sense. Which value of k gives the best distortion for least time.

We need to define more formally, what we mean by optimal choice of k. One intuitive definition is that choose k that gives minimum distortion. Another approach is to

consider the fact that Graph PNN is an approximation of PNN algorithm, find k such that codebook created by Graph PNN is same as created by PNN. On the other hand, it is possible that algorithm cannot reach codebook size M as the number of edges goes to zero before M is reached. Consider the situation, when $k = 1$ and graph is such that all vectors form pairs *i.e.* $e = (\boldsymbol{x}_a, \boldsymbol{x}_b)$ iff $e = (\boldsymbol{x}_b, \boldsymbol{x}_a)$. Then the smallest codebook that can be created is of size $N/2$. It is clear that arbitrary sized codebooks can be created with algorithm only when kNN graph is *connected*. Correspondingly kNN graph must have M *connected components* for the algorithm to be able to reach the target codebook size.

4.5.1 The Case $k = 1$

In the theoretical development of this section we assume that in merge of two clusters S_a and S_b, we do not select k outgoing edges for the new merged, cluster but just take union of outgoing edges of both S_a and S_b. This means that only duplicate edges are removed. Analysis is easier with this assumption. In this section, we want to show that when $k = 1$ is enough for the algorithm to reach target codebook size M. Firts we need the definition of the proximity graph *Minimum Spanning Tree* (MST) [90]. Minimum spanning tree is a spanning tree (tree, which connects all the vertices) with the minimum cost.

Definition 5. *Minimum spanning tree is a tree that spans the vector set S such that the sum of euclidean edge weights is less than the sum for any other spanning tree.*

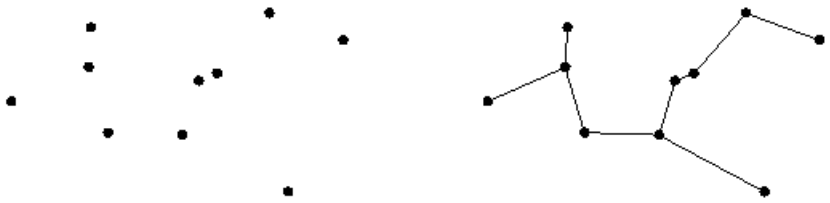


Figure 4.6: Sample $2d$ dataset (left), and the corresponding minimum spanning tree (right) [90].

Now we prove that all edges of the nearest neighbour graph are in minimum spanning tree. We use the ideas from the correctness proof of Prim's algorithm [22].

Lemma 6. *Let $T = (E, V)$ be an minimum spanning tree and $\boldsymbol{q} \in V$, then there exists $e \in E$, where $e = (\boldsymbol{q}, NN(\boldsymbol{q}))$ for all $\boldsymbol{q} \in V$.*

Proof. Let $T' = (E', V')$ be a minimum spanning tree, which does not contain vector $\boldsymbol{q}$ and $\boldsymbol{v} = NN(\boldsymbol{q})$. By definition $d(\boldsymbol{q}, \boldsymbol{v}) \leq d(\boldsymbol{q}, \boldsymbol{x})$, where $\boldsymbol{x} \in V' \setminus \boldsymbol{v}$. Adding $e' = (\boldsymbol{q}, \boldsymbol{x})$ does not create legal minimum spanning tree, as $w(T' + (\boldsymbol{q}, \boldsymbol{v})) \leq w(T' + (\boldsymbol{q}, \boldsymbol{x}))$, where function $w()$ summs all edge weights. □

Lemma 7. $NNG \subseteq MST$

Proof. Proof is a direct application of the lemma 6, as it says that for each vector there exists an edge to its closest neighbour. □

Next we state our main result in this section. We need one more graph theoretical concept, namely *loop*, which is a path of two directed edges such that $e = (a, b)$ and $e' = (b, a)$.

Theorem 1. *Nearest neighbour graph is connected if and only if it has only one loop.*

Proof. "⇒" Let connected nearest neighbour graph be $G = (E', V)$ and minimum spanning tree be $T = (E, V)$. Lemma 6 says that $E' \subseteq E$ and from the properties of the trees we know that $|E| = |V| - 1$ and trivially nearest neighbour graph $|E'| = |V|$. Then one edge in G is left over from the minimum spanning tree and thus forms the loop with one other edge in G.

"⇐" Let's form G' by subtracting the other edge of the loop from the G getting $|E''| = |V| - 1$ edges. To show that G is connected we need to show that $T = G'$. First let's consider the case $G' \subseteq T$. We assume that $e \in G', e \notin T$. But lemma 6 says that $E' \subseteq E$, which leads to contradiction. And lastly we consider the case $T \subseteq G'$. Now as $E'' \supseteq E$ and $|E''| = |E|$, then $E'' = E$. Which proves our claim. □

In practice, this result has a real significance when user wants to use Graph PNN with $k = 1$. By setting up an bookkeeping system to calculate amount of loops after graph creation. By applying the theorem 1, we calculate number of loops then subtract one from the sum, resulting the number of connected components. Example is shown in Fig. 4.7, which contains four connected components and five loops. For this example we could use Graph PNN algorithm to create clusterings where $M \in [4, 14]$. Algorithm will fail if codebook sizes 1, 2 or 3 are requested.

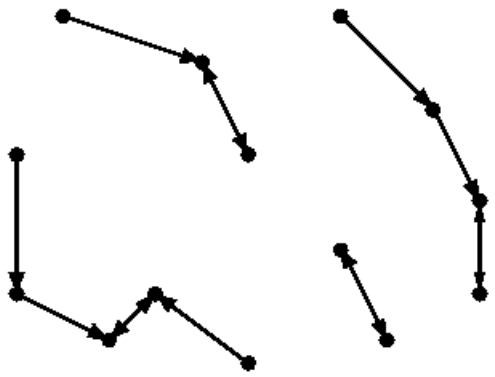


Figure 4.7: Sample $2d$ dataset (left), and the corresponding minimum spanning tree (right) [90].

4.5.2 The Case $k > 1$

In previous section, we discussed on when k could be set to one. We found a fast bookkeeping method to check when clustering can be produced. Unfortunately even though algorithm can produce clustering to the size M, analysis in the previous section does not guarantee anything about the quality of the clustering.

We know from section 4.1.2 that distances between clusters grow monotonously in the PNN iterations. This leads to the observation that the neighbours of cluster do not necessarily stay the same. It is also possible, that the nearest neighbours in the graph creation will stay during the PNN iterations. One example is the case when nearest neighbour pointers form an chain. For that dataset our algorithm will produce exactly same codebook as PNN algorithm with $k = 1$. Both algorithms will successively merge each vector to the same vector.

Optimal selection of the k in terms of PNN algorithm can be stated in the following way. If during the algorithm iterations closest pair is always found in the neighbourhood list, then our algorithm produces exactly same results as the PNN algorithm. Unfortunately no theoretically justified rule of thumb for selecting the k exists. Then good candidate for k can be found experimentally, by comparing the distortion produced by PNN algorithm and our algorithm.

4.6 Creation of the Neighbourhood Graph

Graph creation is related to the post office problem in computational geometry. The problem is to find for every data point $\boldsymbol{x}$ its closest vector from a smaller set of representative vectors. Many solutions exist (e.g. by solving minimum spanning tree for the special case $k = 1$) at the cost of $O(N \log N + M^2)$, and with the assumption that all pairwise distances are stored in memory. In our case, however, we have $M = N$. This makes the time complexity to $O(N^2)$, which is the same as that of the full search.

Efficient construction of the k nearest neighbor graph is needed also in manifold learning [88]. It uses nonlinear dimensionality reduction by mapping the input vectors into a smaller dimensional subspace (called manifold). If such manifold exists, Euclidean distance can be assumed to hold between nearby vectors along the surface, which reduces the dimensionality significantly.

In statistical pattern recognition, the graph creation is known as kNN classifier problem, or k-all-nearest-neighbors problem. The problem is to find for all vectors their k-nearest neighbors in the same set. Theoretical results can be found in [13], but we are not aware of any practical subquadratic algorithm that works for higher dimension D. In general, this is referred as the curse of dimensionality.

Straightforward solution is to construct the graph by *brute force* by considering all pairwise distances $\binom{N}{2}$ but at the cost of $O(N^2)$ time. In the following, we aim at solving the problem either by a faster but heuristic (thus suboptimal) method, or by allowing the worst case time complexity become $O(N^2)$ if the algorithm works faster for typical data sets in practice. We consider also the KD-tree method.

4.6.1 KD-Tree

There are many algorithms using spatial data structures for fast nearest neighbor search such as *KD-tree*, *ball-tree*, and *R-tree*. For example, KD-tree [7] has been introduced already long time ago and it is widely used for finding the nearest neighbor [57]. These techniques can be extended for creating k nearest neighbor graph.

The KD-tree is a generalization of a simple binary search tree, in which each node represents a subset of the vectors in the data set. The *root* of the tree represents the entire data set. Each non-terminal node has two children *sons* representing two subsets defined by the partitioning. The terminal nodes represent mutually exclusive

small subset of the data sets, which collectively form a partition of the data set. These terminal subsets of vectors are called *buckets*. In D-dimensional vector space, a cluster is represented by D keys. Any of these can serve as the discriminator for partitioning the subset represented by a particular node in the tree. In the creation of the neighbourhood graph, we insert every vector into the KD-tree, and then search for each vector its k nearest neighbours from the same tree.

The KD-tree data structure provide an efficient mechanism for examining only those vectors closest to the query vector, thereby greatly reducing the computation required to find the k nearest neighbours. The search algorithm is most easily described as a recursive procedure. The geometric boundaries of the node are determined by the partitions defined at nodes above it in the tree. If the node under investigation is terminal, then all the vectors in the bucket are examined excluding the query vector in question. A list of k nearest neighbours so far found and their distances to the query vector is maintained as an ordered list during the search. Whenever a vector is examined and found to be closer than the most distant member of this list, the list is updated.

If the node under investigation is not terminal, the recursive procedure is called for the node representing the subset on the same side of the partition as the query vector. When the control returns, a test is made to determine if it is necessary to consider the vectors on the side of the partition opposite the query vector. This is referred to as the "*bounds-overlap-ball*" test. A "*ball-with-in-bounds*" test is made before returning to determine if it is necessary to continue the search. See [7] for details of the KD-tree algorithm.

The goal of the optimization of the KD-tree is to minimize the expected number of vectors examined with the search algorithm. The parameters to be adjusted are the discriminating key, partition value at each nonterminal node, and the number of vectors contained in each terminal bucket. The prescription for optimizing the KD-tree is to choose at every nonterminal node the key with the largest spread in values as discriminator and to choose the median of the discriminator key values as the partition.

The creation of the KD-tree takes $O(DN \log N)$ time and each search is proportional to $O(\log N)$ [7]. Thus, if we consider k as a small constant, the expected time complexity of the algorithm is only $O(N \log N)$ in the case of the low dimensional data sets. Nearest neighbour search capabilities of KD-tree has been studied theoretically

and experimentally by Yianilos [111]. It was concluded that for high dimensional uniformly distributed data, to achieve savings over exhaustive search, the search radii has to be very small. In other words, as dimensionality increases either the number of distance computations must be increased, or the search radii decreased.

4.6.2 Other Methods

The other methods that can be considered for searching the k nearest neighbours include:

- *VPT* [111]
- *AESA* [16]
- *MST* [45]
- *TIEC* [68]

Vantage point trees (VPTs) build a binary tree recursively, taking any vector $\boldsymbol{p}$ as the root and taking the *median* M of the set of all distances d. Those vectors $\boldsymbol{u}$ such that $d(\boldsymbol{p},\ \boldsymbol{u}) \leq M$ are inserted into the left subtree, while those such that $d(\boldsymbol{p},\ \boldsymbol{u}) > M$ are inserted into the right subtree. To solve a query in this tree, me measure $d = d(\boldsymbol{q},\ \boldsymbol{p})$. If $d - r \leq M$ we enter into the left subtree, and if $d + r > M$ we enter into the right subtree (with search radius r) . We report every vector considered that is close enough to the query. The VPT takes $O(N)$ space and it is build in $O(N \log N)$ worst case time. The query complexity is argued to be $O(\log N)$, but as pointed out, this is true only for very small search radii, too small to be an interesting case.

Approximation elimination search algorithm (AESA) is experimentally shown to have $O(1)$ query time. The structure is simply a matrix with the $N(N-1)/2$ precomputed distances among the vectors of the data set. At search time a vector $\boldsymbol{p}$ from the data set is selected at random and measure $r_p = d(\boldsymbol{p}, \boldsymbol{q})$, eliminating all vectors u of the data set that do not satisfy $rp - r \leq d(\boldsymbol{u}, \boldsymbol{p}) \leq rp + r$ (with search radius r). While all the $d(\boldsymbol{u}, \boldsymbol{p})$ has been precomputed, so only $d(\boldsymbol{p}, \boldsymbol{q})$ has been calculated at search time. The process of taking a random pivot among the (not yet eliminated) vectors of the data set and eliminating more vectors from the data set is repeated until the candidate set is empty and the query has been satisfied. See [16] for details of the algorithm. The problem with the algorithm is that it needs $O(N^2)$ s pace and construction time which is unacceptably high for all but very small data sets.

Minimum spanning tree (MST) can be constructed in $O(|E|+|V|log|V|)$ time, where $|E|$ is the number of the edges and $|V|$ is the number of the vertices. At each vertex of the MST is a data vector, and the weight of an edge is the distance between the data vectors connected by the edge. In an MST, each vertex is always connected to at least one of its nearest neighbours. Thus, a partial ordering of the data vectors can be obtained. The MST presentation can be precomputed, and then stored in $O(N)$ space. Unfortunately, in the case for searching the k nearest neighbours the precomputing takes $O(N^2 + N \log N)$ time.

The triangle inequality elimination criteria (TIEC) restrict the nearest neighbour search to a subsection of the data set based on the distances of the data vectors to an "anchor" vector. Given a fixed anchor vector, the distances between a anchor and each data vector is pre-computed and stored in $O(N)$ space. Those distances serve as scalar projections of the data vectors with respect to the anchor vector. Using several anchor vectors can strengthen TIEC's ability to eliminate data vectors from consideration. However, each anchor vector requires $O(N)$ space for the scalar projections, and the distances between each anchor vector and the other data vector needs to be calculated. If we use N anchors the TIEC takes $O(N^2)$ space and time.

Chapter 5

Experiments

"When in doubt, print 'em out."
– Karl's Programming Proverb 0x7

In this section, we present our experimental procedure and the results of the subsequent test runs. Testing methodology was as follows: three fairly different natural images were chosen. Images were quantized to 16, 32, 64, 128 and 256 colors with the algorithms described below. Primarily we were interested in the distortion between original image and the quantized image. Distortion function used was MSE, from Eq. (2.12). Finally, execution times of the algorithms were also recorded.

5.1 Experimental Setup

Test images in our experiments are: parrots, sunflower and skyline as shown in Figs. 5.1, 5.2 and 5.3. As in [81], we did not use prequantization. From the test image set, image parrots is a classical image processing and compression test image. Image contains large collection of different colors in the detail areas, on the other hand, background is out of focus and thus is the area of smooth change from green to brown and white. In Fig. 5.1, we see that colors are concentrated on the luminance axis, with a few separate clusters around it. These clusters are connected to the main cluster in the luminance axis.

Sunflower image was taken by the author with Nikon CoolPix 3200 digital camera. The image size was reduced by 60 % to make the dimensions of the image approximately similar to the other test images. From the table 5.1 we see that the image

contains a large amount of unique colors, 47% of the pixels contains mutually different colors. Image contains a high amount of yellow and green tones, also we note that the luminance axis is not so clearly observed as in the parrots image.

Table 5.1: Summary of the test images.

Image	W	H	unique colors	unique/all colors	type
parrots	768	512	72175	18%	Natural (animal)
sunflower	453	604	127282	47%	Natural (flower)
skyline	400	594	47102	20%	Natural (scenery)

Skyline image is the famous desert scenery photo taken in the Arches National Park, Utah[1]. Image contains only relatively few unique colors, which are either different shadesof red and blue. From Fig. 5.3 we can see that the two colors with different shades form two well separated clusters.

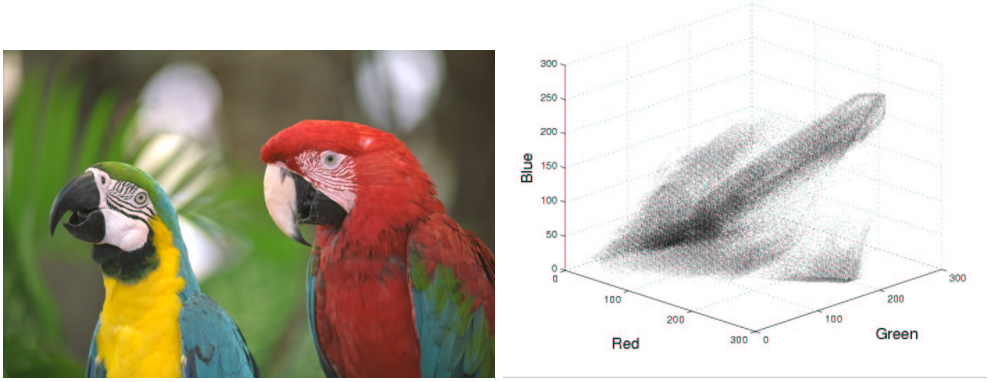


Figure 5.1: Parrots image (left) and the corresponding color histogram (right).

For this study, we selected nine different algorithms. The basis of choosing algorithms on this experiment was to make the comparison between the most common divisive methods, agglomerative methods and the proposed Graph PNN method. The kNN graph was created using KD-tree method described in section 4.6.1. Optimized PNN refers to the implementation of the fast exact PNN algorithm [39] with the speed optimization tricks explained in [95]. For the sake of comparison, popularity algorithm and the generalized Lloyd algorithm were also taken into the test. By Multiple Generalized Lloyd algorithm we mean running GLA on 30 different random initial

[1]Copyright photo courtesy of Robert E. Barber, Barber Nature Photography (REBarber@msn.com).

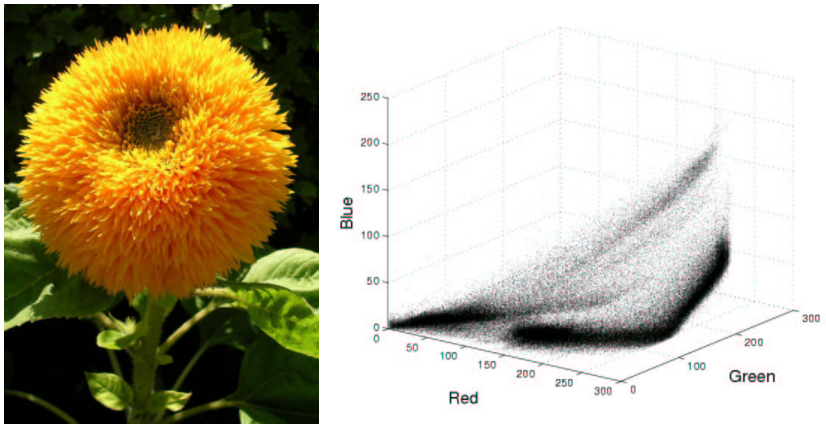


Figure 5.2: Sunflower image (left) and the corresponding color histogram (right).

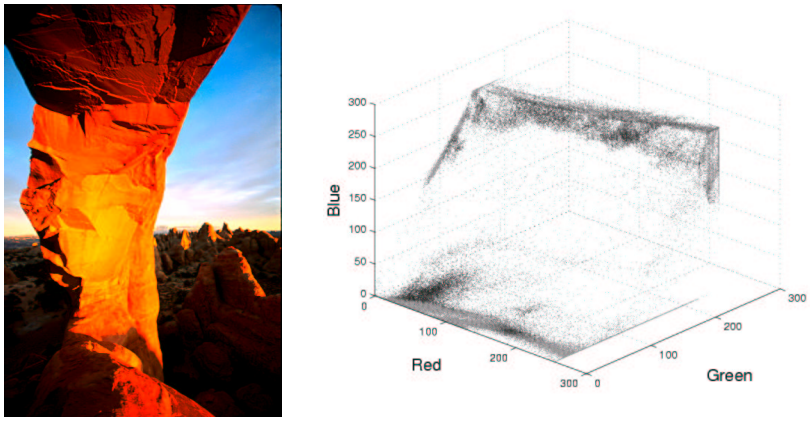


Figure 5.3: Skyline image (left) and the corresponding color histogram (right).

solutions and picking the best solution as the output codebook. We will also study if the codebooks generated from some selected algorithms can be fine-tuned by the GLA.

The list of algorithms used in this experiment are:

- Popularity algorithm [50],
- Generalized Lloyd algorithm [70],
- Multiple Generalized Lloyd algorithm [70],
- Median Cut [50],
- Orchard's algorithm [77],

- Locally Optimal Bi-partitioning [104, 38],
- Pairwise Nearest Neighbour [33],
- Optimized Pairwise Nearest Neighbour [95] and
- Graph PNN [37].

We hypothesize that popularity algorithm will output the codebook with most distortion, compared to the other algorithms we are testing. We see from Figs. 5.1, 5.2 and 5.3 that the colors tend to concentrate on certain areas, then it is plausible that most popular colors are very close to each other. Interesting question is whether GLA can significantly improve the results of the popularity algorithm. In literature, it has been reported that GLA and even running GLA multiple times would not yield as low distortion as agglomerative methods [94, 58]. We would like to know if this result applies also to our test image sets.

In table 5.2, we see eigenvalues calculated from the covariance matrices of the test images. Magnitude of the eigenvalue describes the variance of the data in the direction of the corresponding eigenvector. Both Orchard's algorithm and the locally optimal bi-partitioning assume that the $\lambda_1 > \lambda_2$. For sunflower λ_1 is more than five times of the λ_2, which would suggest that the divisive principal component based methods (Orchard's method and optimal bi-partitioning) work well for the sunflower image.

Table 5.2: Eigenvalues of the test image covariance matrices.

Image	λ_1	λ_2	λ_3
parrots	7069	2286	821
sunflower	9265	1645	340
skyline	1830	604	21

Finally, we are interested to know if agglomerative methods produce better quality color palettes than the divisive methods.

5.2 Results

In the following, we first made the quantization experiments with all algorithms to all test images. We noticed that the popularity algorithm systematically produced distortions of more than thousand. For that reason we used GLA as a fine-tuner of

the popularity results. In tables 5.3, 5.4 and 5.5, we see the quantization results of all algorithms when varying the codebook size. Best distortion for each test is marked with boldface.

We note that for parrots and sunflower, GLA and multiple GLA produces lowest distortions. The reason for this is that when dataset forms essentially one big cluster, centroids in GLA iterations can travel to the locations with low distortion. Hierarchical methods, on the other hand do not iteratively finetune the codebook and even though the individual split or merge decisions are optimal, it does not guarantee global optimum. Median cut produces high distortion in all cases. And also as expected, Orchard's algorithm produces comparatively good results but locally optimal bi-level partitioning produces systematically lower distortion than Orchards algorithm.

Table 5.3: Summary of quantization distortions for parrots.

Algorithm	$M = 16$	$M = 32$	$M = 64$	$M = 128$	$M = 256$
Popularity + GLA	172.13	92.40	58.19	35.47	21.89
GLA	**136.32**	75.39	**41.54**	**24.43**	14.32
Multiple GLA	136.32	**75.08**	41.54	24.43	14.32
Median Cut	640.51	533.81	427.47	310.41	227.15
Orchard's algorithm	146.53	77.18	43.04	24.88	14.63
Optimal Split	137.95	76.32	42.32	24.47	**14.23**
PNN	140.21	78.43	43.76	25.21	14.57
Optimized PNN	141.50	77.91	43.04	25.27	14.53
Graph PNN	138.94	77.55	42.78	24.95	14.50

Table 5.4: Summary of quantization distortions for sunflower.

Algorithm	$M = 16$	$M = 32$	$M = 64$	$M = 128$	$M = 256$
Popularity + GLA	136.97	73.91	50.73	36.83	30.99
GLA	96.63	53.86	32.16	19.71	12.33
Multiple GLA	**96.52**	**53.86**	**32.15**	**19.70**	12.28
Median Cut	302.10	283.26	254.55	31.59	20.43
Orchard's algorithm	99.11	56.88	33.34	20.22	12.31
Optimal Split	98.47	54.89	32.87	19.86	**12.20**
PNN	100.45	56.53	33.36	20.26	12.41
Optimized PNN	102.25	55.73	33.50	20.23	12.50
Graph PNN	99.28	56.01	33.19	20.13	12.40

In table 5.5, we see the quantization results for the skyline test image. As noted before, the skyline dataset consists of two separated clusters. For this test image, GLA does not produce best results and even running GLA multiple times cannot produce much better codebook. Locally optimal bi-level split produces the best results except for $M = 32$, when Graph PNN produces codebook with a little bit smaller distortion.

Table 5.5: Summary of quantization distortions for skyline.

Algorithm	$M = 16$	$M = 32$	$M = 64$	$M = 128$	$M = 256$
Popularity + GLA	131.72	76.15	52.71	20.21	10.53
GLA	117.54	57.07	25.92	16.01	9.70
Multiple GLA	117.54	57.07	25.92	16.01	9.70
Median Cut	873.72	735.73	619.39	478.41	60.44
Orchard's algorithm	118.31	54.37	24.33	11.92	6.34
Optimal Split	**115.93**	52.92	**23.44**	**11.74**	**6.19**
PNN	127.78	53.57	23.67	12.07	6.28
Optimized PNN	121.25	52.56	24.39	12.06	6.28
Graph PNN	117.86	**52.45**	23.63	12.00	6.32

Then we run the test if GLA can be used to finetune the codebooks produced by hierarchical algorithms. We selected only locally optimal bi-partitioning, optimized PNN algorithm and Graph PNN algorithm. We also recorded the total time used in the codebook generation and subsequent GLA iterations. Best time in each test is

marked with italics font. We did not limit the number of iterations, instead we set it to stop until convergence. Tables 5.6, 5.7 and 5.8 summarize the results. We note from the tables that only in one case GLA produces a litte bit better solution than the other methods. We see that plain GLA is slow on all test cases and even optimized PNN produces results faster in three test cases for skyline test image. The reason why GLA used as finetuner is much faster than the plain GLA is that the initial codebook produced by the hierarchical methods was already close to locally optimal solution. In our tests, we noted that at maximum it took 5 seconds to finetune in any of the tests, when plain GLA, for example for sunflower, took over hundred seconds all the cases.

We note that Graph PNN is the fastest method in all but three test cases for skyline, when splitting strategy was fastest. The difference in distortion between locally optimal bi-partitioning and the PNN methods is very small. In most of the cases, the difference in distortion is less than one unit.

Table 5.6: Finetuned quantization results for parrots.

	$M = 16$		$M = 32$		$M = 64$		$M = 128$		$M = 256$	
Algorithm	MSE	Time	MSE	Time	MSE	Time	MSE	Time	MSE	Time
GLA	136.32	61.32	75.39	77.13	41.54	118.82	24.43	186.50	14.32	140.68
Opt. Split	136.29	6.34	**74.11**	7.05	**41.35**	8.51	**23.93**	9.84	**13.95**	12.78
Opt. PNN	**136.18**	176.90	74.79	176.03	41.49	176.93	24.04	177.42	13.97	178.84
Graph PNN	136.37	*5.40*	74.93	*5.34*	41.43	*5.92*	24.17	*6.45*	14.02	*7.68*

Table 5.7: Finetuned quantization results for sunflower.

	$M = 16$		$M = 32$		$M = 64$		$M = 128$		$M = 256$	
Algorithm	MSE	Time	MSE	Time	MSE	Time	MSE	Time	MSE	Time
GLA	**96.63**	140.68	53.86	126.01	32.16	153.62	19.71	220.37	12.33	355.77
Opt. Split	96.92	12.75	54.02	14.16	32.12	16.22	**19.44**	19.09	**11.94**	25.11
Opt. PNN	97.61	447.17	53.87	447.70	32.09	447.85	19.45	449.76	**11.94**	450.83
Graph PNN	96.66	*8.41*	**53.56**	*8.33*	**31.88**	*9.35*	19.47	*10.48*	11.99	*13.07*

Table 5.8: Finetuned quantization results for skyline.

	$M = 16$		$M = 32$		$M = 64$		$M = 128$		$M = 256$	
Algorithm	MSE	Time	MSE	Time	MSE	Time	MSE	Time	MSE	Time
GLA	117.55	28.95	57.10	45.38	25.92	59.05	16.01	96.24	9.70	150.19
Opt. Split	114.77	*2.51*	50.63	*3.10*	**22.79**	*3.77*	**11.53**	4.68	**6.07**	6.72
Opt. PNN	**114.49**	50.81	**50.31**	50.96	23.29	50.48	11.61	50.98	6.08	51.36
Graph PNN	116.76	3.41	51.01	3.62	23.23	3.87	11.67	*4.37*	6.15	*5.21*

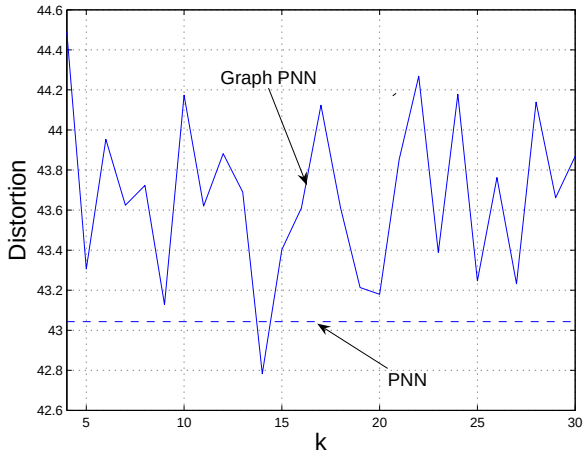
5.2.1 Neighbourhood Size

We studied the effect of the neighbourhood size on the distortion. The results are shown in Figs. 5.4, 5.5 and 5.6 respectively. As can be seen from table, algorithm was not able to reach the target codebook size with $k = 1$. And even more, when target codebook size was 64 and 16, $k = 2$ was not large enough. We note that in all cases, for some value of k, Graph PNN reached the same distortion or smaller than PNN. But as can be seen in Table 5.9, PNN still produces smaller distortion on average (tests were run until $k = 40$) . Standard deviation of the distortion increases noticeably when codebook size is getting smaller.

We note from Fig. 5.4 that when quantizing to 256 colors, only a small neigbourhood is enough. Furthermore, distortion does not change much as the function of k. For that reason, almost any k will produce good results. On the other hand, from these tests it is not possible to give a rule of thumb on how to select the value of k, when codebook size is smaller (64 and 16). This would need further investigations, and possibly more theoretical approach to the problem.

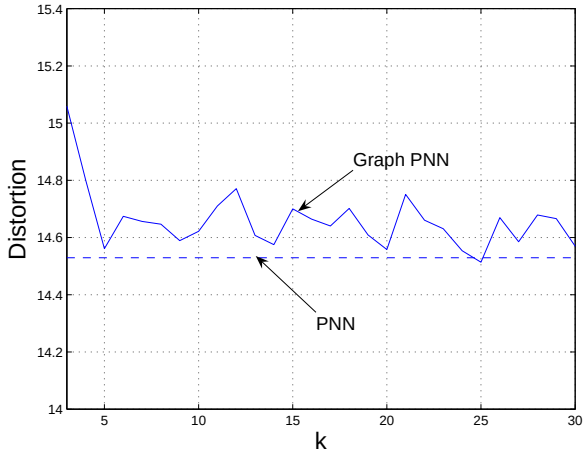
Table 5.9: Summary of distortions.

M	PNN (MSE)	Graph PNN (avg.MSE)	Graph PNN (std.MSE)
16	141.50	144.33	2.79
64	43.04	43.65	0.40
256	14.53	14.63	0.07



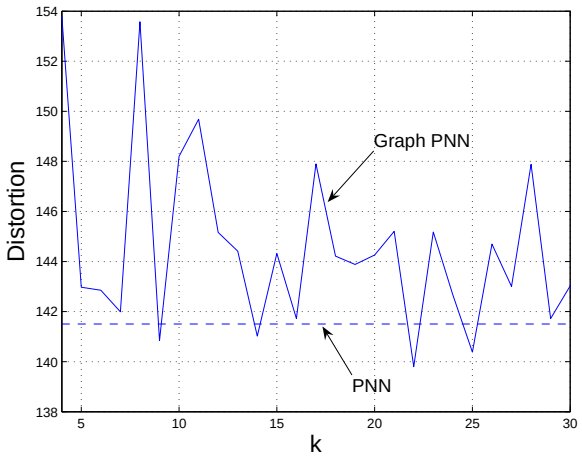
k	Distortion
1	n/a
2	n/a
3	50.22
4	44.49
5	43.31
6	43.95
7	43.62

Figure 5.5: Effect of the neighbourhood size, when quantizing parrots to 64 colors.



k	Distortion
1	n/a
2	52.72
3	15.06
4	14.80
5	14.56
6	14.67
7	14.66

Figure 5.4: Effect of the neighbourhood size, when quantizing parrots to 256 colors.



k	Distortion
1	n/a
2	n/a
3	300.52
4	153.79
5	142.97
6	142.85
7	141.99

Figure 5.6: Effect of the neighbourhood size, when quantizing parrots to 16 colors.

Chapter 6

Conclusions

"A conclusion is the place where you got tired of thinking."
– Anonymous

In this work, we have presented the color quantization framework with emphasis on hierarchical palette generation algorithms. Common palette generation algorithms are reviewed in the first part of this thesis. We presented a review of different divisive heuristics to solve the problem. We find in experiments that divisive methods that project vectors to the principal component are fast and produce a good quality palette compared to the other methods.

Main concentration of this thesis is in the agglomeration methods. Agglomeration method produces good quality codebook, but the algorithm is slow because it considers the full distance matrix from each cluster to all other clusters. Main idea of the fast exact PNN, is that for each cluster only it's nearest neighbour needs to be kept in store. Unfortunately, maintaining the list requires $O(\tau N)$ time. To find the nearest neighbour we have to consider all other clusters. Experimentally we notice also that the fast exact PNN is more than ten times slower than the best locally optimal bi-partitioning method.

We can make agglomeration faster if we approximate the neighbourhood. We do not consider all $N-1$ possible neighbours, but a neighbourhood is calculated before the agglomeration process. In this thesis, we presented an agglomeration algorithm based on the k nearest neighbour graph. We notice in experiments that the algorithm can produce the same quality palette as the locally optimal bi-partitioning and other

agglomeration methods. Graph based agglomeration method is faster than the best splitting method in all but three test cases.

The problem of choosing the best possible size for the neighbourhood was also studied. We proved that for the neighbourhood size of one, if graph has $M+1$ loops, M clusters can be formed. We studied also experimentally how the distortion was affected by the function of neighbourhood size. We noticed that when $M = 256$ neighbourhood size can be arbitrary as long as it is not one or two. But with smaller palette size we noticed a lot of variation in distortion. It is still an open theoretical problem, how to select the best possible k.

Appendix A

Principal Component Analysis

"Many students feel that mathematics is an old, dead subject. And why not? Most of high school mathematics was perfected many centuries ago by the Greeks and Arabs, or at the latest, a few centuries ago by Newton and Leibnitz."
– Michael Frame and Benoit Mandelbrot

A.1 Covariance Matrix Calculation

To calculate all principal components of the color space we need to first center the dataset around origin by subtracing the mean value:

$$\hat{\boldsymbol{x}}_i = \boldsymbol{x}_i - \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{x}_j, \tag{A.1}$$

and we form a matrix of all subtracted vectors:

$$\boldsymbol{X} = (\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_n). \tag{A.2}$$

Then we calculate *covariance matrix* $\boldsymbol{\Sigma}$:

$$\boldsymbol{\Sigma} = \frac{1}{n-1}\boldsymbol{X}\boldsymbol{X}^T, \tag{A.3}$$

which can be justified as *unbiased* covariance of two sample values defined as:

$$\sigma_{ij} = \frac{1}{n-1}\sum_{k=1}^{n}(x_{ki} - \bar{x}_i)(x_{kj} - \bar{x}_j). \tag{A.4}$$

This means that covariance σ_{ij} is a variance measured between dimensions i and j. As the individual elements in matrix X from Equation A.3 were already mean

subtracted we can see identity between Equation A.4 and Equation A.3 when matrix $\mathbf{\Sigma}$ in Equation A.3 is expanded:

$$\sigma_{ij} = \sum_{k=1}^{n} x_{ik} x_{jk}. \tag{A.5}$$

A.2 Eigenvectors and Eigenvalues of Covariance Matrix

Eigenvector $\boldsymbol{v}_1$ that corresponds to the largest eigenvalue λ_1 of covariance matrix $\mathbf{\Sigma}$ descibes the direction of largest variance in data [49]. That eigenvector is also called first principal component. Similarly eigenvector $\boldsymbol{v}_2$ that corresponds to the second largest eigenvalue λ_2 is called second principal component of data and so on.

Eigenvector and it's corresponding eigenvalue of covariance matrix $\mathbf{\Sigma}$ is defined:

$$\mathbf{\Sigma}\boldsymbol{v} = \lambda \boldsymbol{v}, \tag{A.6}$$

where we get:

$$(\mathbf{\Sigma} - \lambda \boldsymbol{I})\boldsymbol{v} = \mathbf{0}. \tag{A.7}$$

To which trivial solution is $\boldsymbol{v} = \mathbf{0}$, and non-trivial solutions are found by using *Invertible Matrix Theorem* and rewriting Eq. A.7 to:

$$\det(\mathbf{\Sigma} - \lambda \boldsymbol{I}) = 0. \tag{A.8}$$

This leads to solving the roots of the d degree polynomial:

$$p(\lambda) = c_0 \lambda^d + c_1 \lambda^{d-1} + \ldots + c_{d-1}\lambda + c_d \tag{A.9}$$

The maximum number of roots of d degree polynomial is d, and thus, the number of eigenvectors in $\mathbf{\Sigma}$ is d. Every $\boldsymbol{x}$ of $\mathbf{\Sigma}$ is uniquely represented by the linear combination of $\boldsymbol{v}_i$'s:

$$\boldsymbol{x} = c_1 \boldsymbol{v}_1 + c_2 \boldsymbol{v}_2 + \ldots c_d \boldsymbol{v}_d. \tag{A.10}$$

So it can be seen that the eigenvectors form a *basis* on $\mathbf{\Sigma}$.

Then problem is to find those eigenvectors. Whatever algorithm we use, we can use the definition of eigenvector and eigenvalue in Eq. A.6 to check whether the result was correct, or close enough to be usefull in practice. Solving the roots of the polynomial with a high degree is not practical. Thus algorithms are iterative that approach desired eigenvector and corresponding eigenvalue. These algorithms provably eventually find correct eigenvector.

A.2.1 Power Method

Power method is an iterative method for finding largest eigenvector and corresponding eigenvalue of matrix $\mathbf{\Sigma}$. First we have to assume that relationship

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_d| \tag{A.11}$$

holds. This means that $|\lambda_1|$ is *strictly larger* than $|\lambda_2|$. If this assumption of data does not hold, then power method will not converge. Correspondingly closer $\frac{|\lambda_2|}{|\lambda_1|}$ gets to 1 slower power method converges.

Algorithm 8 Power Method

Select $\boldsymbol{x}_0$, whose largest element is 1.
repeat
  $\hat{\boldsymbol{x}}_{k+1} \leftarrow \Sigma \boldsymbol{x}_k$
  $\mu_k \leftarrow \max\{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_d\}$
  $\boldsymbol{x}_{k+1} = \frac{\hat{\boldsymbol{x}}_{k+1}}{\mu_k}$
  $k \leftarrow k + 1$
until $\boldsymbol{x}$ converges

Power method is an iterative algorithm, described in Algorithm 8, which starts from initial guess eigenvector $\boldsymbol{x}_0$. New vector $\boldsymbol{x}_{k+1}$ is calculated by multiplying it by covariance matrix Σ and then scaling vector component values by maximum component value μ_k. The sequence $\{\mu_k\}$ approaches λ_1 and $\boldsymbol{x}_k$ approaches $\boldsymbol{v}_1$ as $k \to \infty$.

Fig. A.1 is an example of how sequence $\{\mu_k\}$ reaches λ_1 with just a few iterations. Covariance matrix calculated from the parrots test images has a $\lambda_1 = 6323$. Power method reaches 6314 with just five iterations and with twelve iterations reaches correct the eigenvalue.

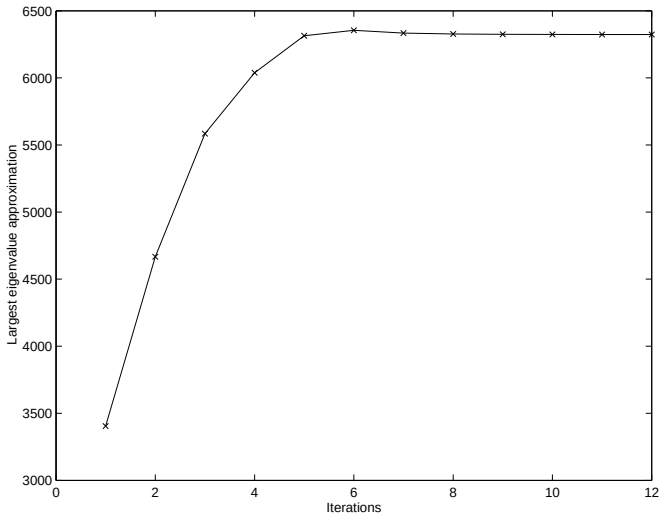


Figure A.1: Example of convergence of $\{\mu_k\}$ sequence with Σ from parrots test image.

Problem with the power method is that it can only be used to approximate largest eigenvalue and its corresponding eigenvector.

Bibliography

[1] A. Akimov, A. Kolesnikov, and P. Fränti. Coordinate quantization in vector map compression. In *Proceedigs IASTED International Conference on Visualization, Imaging and Image Processing (VIIP'04)*, pages 748–753, Marbella, Spain, September 2004.

[2] S. Arya and D. M. Mount. *The Handbook of Data Structures and Applications*, chapter Computational Geometry: Proximity and Location. CRC Press LLC, Boca Raton, Florida, 2005.

[3] R. Balasubramanian and J. P. Allebach. A new approach to palette selection for color images. *The Journal of Imaging Science and Technology*, 17(6):284–290, December 1991.

[4] R. Balasubramanian, C. A. Bouman, and J. P. Allebach. Sequential scalar quantization of color images. *Journal of Electronic Imaging*, 3:45–59, January 1994.

[5] G. H. Ball and D. J. Hall. A clustering technique for summarizing multivariate data. *Behavioral Science*, 12(2):153–155, March 1967.

[6] R. Bellman. *Dynamic programming*. Princeton University Press, New Jersey, 1957.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[8] J. Boberg and T. Salakoski. General formulation and evaluation of agglomerative clustering methods with metric and non-metric distances. *Pattern Recognition*, 26(9):1395–1406, September 1993.

[9] G. W. Braudaway. A procedure for optimum choice of a small number of colors from a large color palette for color imaging. In *Proceedings of Electronic Imaging '86 - International Electronic Imaging Exposition and Conference*, pages 75–79, Boston, November 1986.

[10] M. R. Brito, E. L. Chávez, A. J. Quiroz, and J. E. Yukich. Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35(1):33–42, August 1997.

[11] J. D. Bruce. Optimum quantization. Technical Report 429, MIT R.L.E., 1965.

[12] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel. Speech coding based upon vector quantization. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(5):562–574, October 1980.

[13] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the Association for Computing Machinery*, 42(1):67–90, 1995.

[14] J. Cardinal and D. Eppstein. Lazy algorithms for dynamic closest pair with arbitrary distance measures. In *SIAM Workshop on Algorithm Engineering and Experiments (ALENEX04)*, pages 112–119, New Orleans, January 2004.

[15] N. Chaddha, W.-C. Tan, and T. H. Y. Meng. Color quantization of images based on human vision perception. In *Proc. IEEE Acoustics, Speech, and Signal Processing*, volume 5, pages 89–92, Adelaide, Australia, April 1994.

[16] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[17] H. Cheng. *Document Image Segmentation and Compression.* PhD thesis, Purdue University, August 1999.

[18] B. T. Chun, Y. Bae, and T.-Y. Kim. Caption segmentation methodin videos using isodata clustering of topological features. In *Proceedings of the IEEE Region 10 Conference*, pages 915–918, Cheju Island, South Korea, September 1999.

[19] B. T. Chun, J. Soh, and J. H. Yoo. A hybrid thresholding method for degraded vehicle number plate images. *Journal of the Korean Institute of Telematics and Electronics*, 31-B(10):112–122, 1994.

[20] CIE. *Commission Internationale de l'Eclairage Proceedings*. Cambridge University Press, Cambridge, United Kingdom, 1931.

[21] J. Conway and N. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, New York, 1998.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Intoduction to Algorithms*. The MIT Press, 1998.

[23] T. N. Cornsweet. *Visual Perception*. Academic Press, New York, 1970.

[24] T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, 14:326–334, 1965.

[25] C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, and D. H. Salesin. Computer generated watercolor. In *In Proceedings of SIGGRAPH 97*, pages 421–430, 1997.

[26] F. d'Amore and P. G. Franciosa. On the optimal binary plane partition for sets of isethetic rectangles. *Information Processing Letters*, 44(5):255–259, December 1992.

[27] M. de Berg, M. van Kreveld M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer Verlag, New York, second edition, 2000.

[28] S. S. Dixit. Quantization of color images for display/printing on limited color output devices. *Computers and Graphics*, 15(4):561–567, 1993.

[29] A. M. e Sá. Quantizacão de cores em imagens. Master's thesis, Matemática Aplicada - IM/UFRJ, Rio de Janeiro, Brazil, April 2001.

[30] A. M. e Sá and P. C. P. Carvalho. Color quantization by pairwise clustering using a reduced graph. In *Electronic Notes in Discrete Mathematics*, volume 7. Elsevier, 2001.

[31] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pair. *The ACM Journal of Experimental Algorithmics*, 5(1):1–23, June 2000.

[32] D. Eppstein, M. S. Paterson, and F. F. Yao. On nearest-neighbor graphs. *Discrete and Computational Geometry*, 17:263–282, 1997.

[33] W. H. Equitz. A new vector quantization clustering algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(10):1568–1575, October 1989.

[34] R. W. Floyd and L. Steinberg. An adaptive algorithm for spatial gray scale. *Symposium of Technical Papers (SID)*, pages 36–37, 1975.

[35] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading (MA), 2nd edition, July 1997.

[36] P. Fränti and V. Hautamäki. Compression of aerial images for reduced-color devices. In *SPIE Conference on Image and Video Communications and Processing*, volume 5022, Santa Clara, USA, January 2003.

[37] P. Fränti, O. Virmajoki, and V. Hautamäki. Graph-based agglomerative clustering. In *Proceedings of The Third IEEE Int. Conf. on Data Mining*, pages 525–528, Melbourne, Florida, November 2003.

[38] P. Fränti, T. Kaukoranta, and O. Nevalainen. On the splitting method for vector quantization codebook generation. *Optical Engineering*, 36(11):3043–3051, November 1997.

[39] P. Fränti, T. Kaukoranta, D. F. Shen, and K.-S. Chang. Fast and memory efficient implementation of the exact pnn. *IEEE Transactions on Image Processing*, 9(5):358–369, 2000.

[40] R. S. Gentile, J. P. Allebach, and E. Walowit. Quantization of color images based on uniform color spaces. *Journal of Imaging Technology*, 16(11-21), 1990.

[41] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publisher, Boston, USA, 1992.

[42] M. Gervautz and W. Purgathofer. A simple method for color quantization: Octree quantization. In A. S. Glassner, editor, *Graphics Gems I*, pages 287–293. Academic Press, New York, 1990.

[43] A. S. Glassner, editor. *Graphics Gems I.* Academic Press, San Diego, June 1990.

[44] R. C. Gonzalez and R. E. Woods. *Digital Image Processing.* Prentice Hall, Upper Saddle River (NJ), 2nd edition, 2002.

[45] J. C. Gover and G. J. S. Ross. Minimum spanning trees and single linkage clustering analysis. *Applied Statistics*, 18:54–64, 1969.

[46] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, October 1998.

[47] A. Habibi and P. A. Wintz. Image coding by linear transformation and block quantization. *IEEE Transactions in Communications*, 19:50–61, February 1971.

[48] V. Hautamäki, I. Kärkkäinen, and P. Fränti. Outlier detection using k-nearest neighbour graph. In *17th International Conference on Pattern Recognition (ICPR 2004)*, pages 430–433, Cambridge, United Kingdom, August 2004.

[49] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, July 1998.

[50] P. Heckbert. Color image quantization for frame buffer display. In *Proceedings of SIGGRAPH*, volume 16, pages 297–307, Boston, July 1982.

[51] P. Howard, F. Kossentini, B. Martins, S. Forchhammer, and W. Rucklidge. The emerging JBIG2 standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(7):838–848, November 1998.

[52] H. Ing-Sheen and F. Kuo-Chin. An adaptive clustering algorithm for color quantization. *Pattern Recognition Letters*, 21:337–346, 2000.

[53] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[54] J. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, September 1992.

[55] G. Joy and Z. Xiang. Center-cut for color-image quantization. *The Visual Computer*, 10(1):62–66, 1993.

[56] G. Joy and Z. Xiang. Reducing false contours in quantized color images. *Computer and Graphics*, 20(2):231–242, March 1996.

[57] T. Kanungo, D. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[58] T. Kaukoranta. *Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization.* PhD thesis, University of Turku, Turku, Finland, December 1999.

[59] T. Kaukoranta, P. Fränti, and O. Nevalainen. Vector quantization by lazy pairwise nearest neighbor method. *Optical Engineering*, 38(11):1862–1868, November 1999.

[60] K. M. Kim, C. S. Öee, E. joo Lee, and Y. H. Ha. Color image quantization using weighted distortion measure of HVS color activity. In *International Conference on Image Processing (ICIP)*, volume 3, pages 1035–1039, Switzerland, 1996.

[61] P.-K. Kim. Automatic text location in complex color images using local color quantization. In *Proceedings of the IEEE Region 10 Conference*, pages 629–632, Jeju Island, South Korea, September 1999.

[62] D. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, Reading, Massachusetts, March 1975.

[63] P. Kopylov and P. Fränti. Color quantization of map images. In *IASTED Conference on Visualization, Imaging, and Image Processing (VIIP'04)*, pages 837–842, Marbella, Spain, September 2004.

[64] T. Kurita. An efficient agglomorative clustering algorithm using a heap. *Pattern Recognition*, 25(3):205–209, March 1991.

[65] M.-C. Larabi, N. Richard, and C. Fernandez. A fast color quantization using a matrix of local pallets. In *Proc. 29th Applied Imagery Pattern Recognition Workshop*, pages 136–140, Washington DC, USA, October 2000.

[66] H. V. Le. Image processing and analysis for melanoma diagnosis. Master's thesis, California State Polytechnic University, 1999.

[67] K. Lemström and P. Fränti. N-candidate methods for location invariant dithering of color images. *Image and Vision Computing*, 18(6–7):493–500, 2000.

[68] W. Li and E. Salari. A fast vector quantization encoding method for image compression. *IEEE Transactions on Circuit and Systems for Video Technology*, 5(2):119–123, April 1995.

[69] X. Li, T. Yuan, N. Yu, and Y. Yuan. Adaptive color quantization based on perceptive edge protection. *Pattern Recognition Letter*, 24(16):3165–3176, December 2003.

[70] Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer desing. *IEEE Transactions on Communications*, 28(1):84–95, January 1980.

[71] T.-S. Liu and L.-W. Chang. Greedy tree growing for color image quantization. In *Preceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages 97–100, Adelaide, Australia, April 1994.

[72] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):192–137, 1982.

[73] D. S. J. M. R. Garey and H. S. Witsenhausen. The complexity of the generalized Lloyd-Max problem. *Transactions on Information Theory*, 28(2):255–256, 1982.

[74] J. B. McQueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, 1967.

[75] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM*, 44(1):1–29, 1997.

[76] N. B. Nill. A visual model weighted cosine transform for image compression and quality assessment. *IEEE Transactions on Communications*, 33(6), June 1985.

[77] M. T. Orchard and C. A. Bouman. Color quantization of images. *IEEE Transactions on Signal Processing*, 39(12):2677–2690, December 1991.

[78] W. Pennebaker and J. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[79] J. Puzicha, M. Held, J. Ketterer, J. M. Buhmann, and D. W. Fellner. On spatial quantization of color images. *IEEE Transactions on Image Processing*, 9(4):666–682, April 2000.

[80] S. Redfield and J. G. Harris. The role of massive color quantization in object recognition. In *International Conference on Image Processing (ICIP)*, pages 137–186, Vancouver, Canada, September 2000.

[81] P. J. Reitan. *Hybrid approaches to color image quantization*. PhD thesis, University of Maryland, August 1999.

[82] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[83] J. Shanbehzadeh and P. O. Ogunbona. On the computational complexity of the lbg and pnn algorithms. *IEEE Transactions on Image Processing*, 6:614–616, 1997.

[84] J. A. Shufelt. Color image quantization enhancement techniques. Technical report, Carnegie Mellon University: Computer Science Department, September 1995.

[85] C. Shyi-Chyi and Y. Chen-Kuei. A fast and novel techique for color quantization using reduction of color space dimensionality. *Pattern Recognition Letters*, 22:845–856, 2001.

[86] P. H. A. Sneath and R. R. Sokal. *Numerical Taxonomy*. W. H. Freeman & Co, San Francisco, 1973.

[87] D. S. Taubman and M. W. Marcellin. *Jpeg2000: Image Compression Fundamentals, Standards, and Practice*. Kluwer Academic Publishers, 2001.

[88] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000.

[89] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Academic Press, San Diego, 2nd edition edition, February 2003.

[90] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[91] A. Vasilache. *On Vector Quantization with Regular Geometric Structure*. PhD thesis, Tampere University of Technology, 2003.

[92] L. Velho, J. Gomez, M. Vinicius, and R. Sobreiro. Color image quantization by pairwise clustering. In *Proc. of the Brazilian Symposium of Computer Graphics and Image Processing (SIBGRAPI97)*, pages 596–601, Campos do Jordão, Brazil, October 1997.

[93] O. Verevka and J. W. Buchanan. Local k-means algorithm for colour image quantization. In *Proceedings of Graphics Interface '95*, pages 128–135, Québec, Canada, May 1995.

[94] O. Virmajoki. *Pairwise Nearest Neighbor Method Revisited*. PhD thesis, University of Joensuu, Joensuu, Finland, December 2004.

[95] O. Virmajoki, P. Fränti, and T. Kaukoranta. Practical methods for speeding-up the pairwise nearest neighbor method. *Optical Engineering*, 40(11), November 2001.

[96] S. J. Wan, P. Prusinkiewicz, and S. K. M. Wong. Variance-based color image quantization for frame-buffer display. *Color Research and Application*, 15:52–58, Febrary 1990.

[97] S. J. Wan, S. K. M. Wong, and P. Prusinkiewicz. An algorithm for multidimensional data clustering. *ACM Transactions on Mathematical Software (TOMS)*, 14(2):153–162, 1988.

[98] B. A. Wandell. *Foundations of Vision*. Sinauer Associates Inc., Sunderland, Massachusetts, USA, 1995.

[99] J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of American Statistical Association*, 58(301):236–244, March 1963.

[100] X. Wu. Optimal quantization by matrix searching. *Journal of Algorithms*, 12(4):663 – 673, December 1991.

[101] X. Wu. Color quantization by dynamic programming and principal analysis. *ACM Transactions on Graphics*, 11(4):348–372, October 1992.

[102] X. Wu. YIQ vector quantization in a new color palette architecture. *IEEE Transactions on Image Processing*, 5(2):321–329, February 1996.

[103] X. Wu and I. H. Witten. A fast k-means type clustering algorithm. Technical report, Department of Computer Science, The University of Calgary, June 1985.

[104] X. Wu and K. Zhang. A better tree-structured vector quantizer. In *Proceedings Data Compression Conference*, pages 392–401, Snowbird, April 1991.

[105] X. Wu and K. Zhang. Quantizer monotonicities and quantizer design algorithms. *IEEE Transactions on Information Theory*, 39(3):1049–1053, May 1993.

[106] Y. Wu and D. C. Coll. Single bit-map block truncation coding of color images. *IEEE Journal of Selected Areas in Communications*, 10(5):952–959, 1992.

[107] Z. Xiang and G. Joy. Color image quantization by agglomerative clustering. *IEEE Computer Graphics and Applications*, pages 44–48, 1994.

[108] C.-Y. Yang and J.-C. Lin. Use of radius weighted mean to cluster two-class data. *Electronics Letters*, 30(10):757–759, 1994.

[109] C.-Y. Yang and J.-C. Lin. Color quantization by RWM-cut. In *Proceedings of the Third Int. Conf. on Document Analysis and Recognition*, volume 2, pages 669–672, Montreal, August 1995.

[110] J.-W. Yeh and M. Ouhyoung. Non-photorealistic rendering in chinese painting of animals. *Journal of System Simulation*, 14(6):1220–1226, 2002.

[111] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *(SODA) Symposium on Discrete Algorithms*, 1993.

[112] M. P. Yu and M. C. Lo. Contextual algorithm for color quantization. *Journal of Electronic Imaging*, 12(3):442–447, July 2003.