Efficient Multilevel Image Thresholding

Thesis submitted
in partial fulfillment of the requirement for the degree of
Dipl. Ing. FH

Hochschule für Technik Rapperswil

Authors:

Marco Eichmann, Martin Lüssi

Advisors:

Prof. Dr. Aggelos K. Katsaggelos
Prof. Dr. Guido M. Schuster

Rapperswil, December 2005

Authors:	Marco Eichmann[1]	eichmann@gmail.com
	Martin Lüssi[1]	mluessi@gmail.com
Advisors:	Prof. Dr. Aggelos K. Katsaggelos[2]	aggk@ece.northwestern.edu
	Prof. Dr. Guido M. Schuster[1]	guido.schuster@hsr.ch

[1]Hochschule für Technik Rapperswil, Switzerland

[2]Department of Electrical Engineering and Computer Science, Northwestern University, Evanston USA

LaTeX 2_ε

Abstract

Thresholding is one of the most widely used image segmentation operations; one application is foreground-background separation. Multilevel thresholding is the extension to segmentation into more than two classes. In order to find the thresholds, which separate the classes, the histogram of the image is analyzed. In most cases, the optimal thresholds are found by the minimazing or maximazing an objective function, which depends on the positions of the thresholds. We identify a class of objective functions for which the optimal thresholds can be found using algorithms with low time complexities. We also show, that two well known objective functions are members of this class. By implementing the algorithms and comparing their execution times, we can make a quantitative statement about their performance.

Acknowledgements

We gratefully thank Professor Guido M. Schuster and Professor Aggelos K. Katsaggelos for giving us the great opportunity to write our diploma thesis at Northwestern University. Our special thank goes to Professor Aggelos K. Katsaggelos, for his hospitality and for his support.

We would also like to thank Professor David L. Neuhoff for finding time to join our meetings and for the stimulating discussions.

We are also indebted to all the members of IVPL and other students from Northwestern University for their friendship and for makeing our time in Evanston a great experience.

Marco Eichmann Martin Lüssi

Contents

1 Introduction

Thresholding is a very low-level image segmentation technique. It is widely used as a preliminary step, to separate object(s) and background. The principal idea is, that the intensity values of object pixels and the background pixels differ, such that object and background can be separated by selecting an appropriate threshold. In Multilevel Image Thresholding more than one thresholds are set, which segments the image into several classes.

Figure 2.1 shows one example, where a medical image is segmented into three classes, by setting two thresholds t_1 and t_2. The representation of the segmented image is depending on the application, and is not part of this thesis. In this example all pixels with intensity level lower or equal to t_1, belong to class one and are represented by the value 0. Pixels in class two ($t_1 < g \leq t_2$), are represented with the mean intensity of class two, and pixels in class three are shown with intensity value 255.

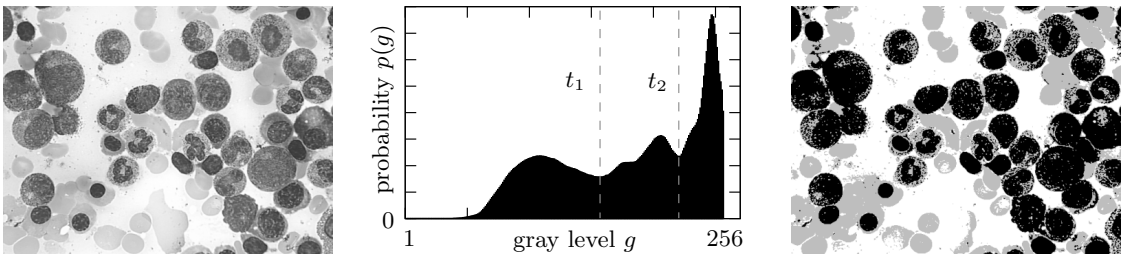

Figure 1.1: Original image, histogram and segmented image.

In the last three decades numerous methods have been proposed, which set the thresholds according to a certain criterion, an overview can be found in [1]. In this thesis, generic algorithms are studied, which can be employed to find the optimal thresholds efficiently. Just thresholding techniques, which employ the gray scale histogram to find the optimal thresholds, are taken into account. As a result, two classes of objective functions are identified. For the first class, an efficient dynamic programming (DP) algorithm can be used for finding the thresholds, whereas for the second class a combination of dynamic programming and fast matrix searching can be employed. Furthermore, it is shown that some well known thresholding techniques are members of these classes. To verify the efficiency of the algorithms, runtime measurements of ANSI C implementations are presented. As an independent topic, the problem about how many classes are present in an image, and how to automatically find this number, is addressed.

This thesis is organized as follows: In Chapter 2, the main problem of multilevel image thresholding is identified in a general matter. For objective functions with a certain structure, a dynamic programming approach is presented in Chapter 3. Furthermore, it is shown in Chapter 4, that if the objective function has useful mathematical properties, sophisticated and efficient matrix searching techniques can be used to further improve

the speedup achieved with dynamic programming. In Chapter 5 it is shown, that for some of the known thresholding methods surveyed in [1], the presented algorithms can be employed. Some details of the C implementations are discussed in Chapter 6, and a quantitative statement about their performance is made in Chapter 7. The problem of the automatic determination of the best number of classes is addressed in Chapter 8. Conclusions are drawn in Chapter 9.

2 Problem Formulation

In this chapter the main problems of multilevel thresholding are identified. Further, a unified notation is introduced which is used throughout this thesis.

The pixels of a observed image are represented in L gray levels g from $1 \dots L$. Multilevel image thresholding is the task of separating the pixels of the image in M classes $C_1 \dots C_M$, by setting the thresholds $t_1 \dots t_{M-1}$. Therefore C_1 contains all pixels with gray levels

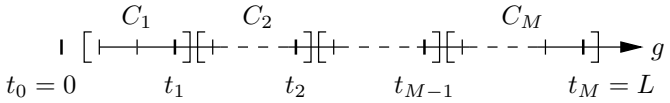

Figure 2.1: Separation of gray levels into classes.

$t_0 < g \leq t_1$, class C_2 all pixels in the range $t_1 < g \leq t_2$ and so on. Note that the highest gray level $g = L$ is always in class C_M. The thresholds t_0 and t_M are not evaluated, they are defined to be 0 and L, respectively.

For the placement of the thresholds most of the thresholding algorithms employ the histogram $h(g)$. The histogram is a statistic of the image and $h(i)$ shows the occurrence of gray level i, where $\sum_{i=1}^{L} h(i) = N$ and N is the number of image pixels. The normalized histogram $p(i)$ can be considered as the probability mass function of the gray levels present in the image.

$$p(i) = h(i)/N\,, \qquad \sum_{i=1}^{L} p(i) = 1. \tag{2.1}$$

For all classes, statistical properties such as the probability of the class (referred as the class weight), the mean or the variance of the class can be calculated as follows.

$$\begin{aligned} \text{class weight}: \qquad w_k &= \sum_{i \in C_k} p(i), \\ w(t_{k-1}, t_k] &= \sum_{i=t_{k-1}+1}^{t_k} p(i). \end{aligned} \tag{2.2}$$

$$\begin{aligned} \text{class mean}: \qquad \mu_k &= \sum_{i \in C_k} p(i) \cdot i / w_k, \\ \mu(t_{k-1}, t_k] &= \sum_{i=t_{k-1}+1}^{t_k} p(i) \cdot i / w_k. \end{aligned} \tag{2.3}$$

$$\text{class variance}: \qquad \sigma_k^2 = \sum_{i \in C_k} p(i) \cdot (i - \mu_k)^2 / w_k,$$

$$\sigma^2(t_{k-1}, t_k] = \sum_{i=t_{k-1}+1}^{t_k} p(i) \cdot (i - \mu_k)^2 / w_k. \tag{2.4}$$

In this thesis both, the class notation (e.g. w_k) and the interval notation (e.g. $w(t_{k-1}, t_k]$) are used, depending on which one suits better.

Thresholding methods, which just analyze the histogram, are usually very simple and efficient and are therefore suitable for the use in real time systems. More sophisticated methods, which also consider spatial information, are not discussed in this thesis. Furthermore, just methods which find the thresholds by minimizing or maximizing a certain criterion are analyzed. This criterion is referred to as objective function.

2.1 Objective Function

The objective function is the central part of the thresholding methods considered in this thesis. The value of the objective function dependens on the positions of the thresholds. The the optimal thresholds are found, by either minimizing or maximizing the objective function. Therefore, all methods discussed further on find the optimal thresholds as follows:

$$[t_1^*, t_2^*, \ldots, t_{M-1}^*] = \underset{0<t_1<\ldots<t_{M-1}<L}{\arg\min} \left\{ J_{M,L}(t_1, \ldots, t_{M-1}) \right\}, \tag{2.5}$$

or

$$[t_1^*, t_2^*, \ldots, t_{M-1}^*] = \underset{0<t_1<\ldots<t_{M-1}<L}{\arg\max} \left\{ J_{M,L}(t_1, \ldots, t_{M-1}) \right\}, \tag{2.6}$$

where $J_{M,L}(t_1, \ldots, t_{M-1})$ is the objective functions.

2.2 Exhaustive Search

For a given number of classes M, the main task in multilevel thresholding is to find the positions of the $M-1$ thresholds, which minimize or maximize the objective function. The obvious and straightforward solution to this problem is to calculate the objective function of every possible placement of the thresholds and take the positions for which the objective function is optimal. This approach is called exhaustive search and encounters the problem of trying every possible placement of thresholds within given boundaries, which is knows as a combination problem. Equation (2.7) gives the number of ways of picking k unordered outcomes from n possibilities, also known as binomial coefficient.

The Figure 2.2 shows an example for the case, where the number of gray levels is $L = 5$ and the number of classes is $M = 3$. In this example $M - 1 = 2$ thresholds have to be set. With the restriction that the last gray level has to be in C_M, the thresholds can only be placed in the interval $[1 \ldots 4]$. In this simple example the objective function has to be calculated $\binom{L-1}{M-1} = 6$ times. But for more realistic examples, where an image has $L = 256$ gray levels and the number of classes is $M = 5$, the number of times the objective function has to be calculated is $\binom{255}{4} = 172'061'505$. For real time implementations the exhaustive search is therefore not a solution and faster algorithms, which find the optimal thresholds without checking every possible placement, are needed.

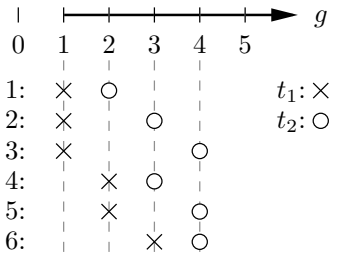

Figure 2.2: Example for threshold placement.

Combination:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{2.7}$$

$$\Rightarrow \binom{L-1}{M-1} = \binom{4}{2} = \frac{4 \cdot 3 \cdot 2 \cdot 1}{2 \cdot 1 \cdot 2 \cdot 1} = 6$$

Dynamic programming (DP), a well known technique for solving such problems, is introduced in the next section.

3 Dynamic Programming Approach

Dynamic programming (DP) is a well known and generic technique for solving optimization problems, where the term "programming" in this context does not refer to writing computer code. Dynamic programming breaks the problem into subproblems, finds the solution to each subproblem, and obtains the overall solution by combining the solutions of the subproblems. More informations about dynamic programming can be found in [2].

For a class of objective functions with a certain structure, an efficient DP algorithm, known as the shortest path algorithm, can be employed to find the optimal thresholds with $O(ML^2)$ time complexity. In this chapter, the required structure is presented and the algorithm is explained. Later in this chapter, the time complexity of this DP algorithm is derived. In order to reduce redundancy, the derivations are just shown for the case where the objective functions is maximized. Since an objective fuction which is minimized can be converted in one which is maximized by simply setting a negative sign, it is obvious, that the following algorithms can be used for both cases.

The shortest path algorithm can be employed, if the objective function $J_{M,L}(t_1, \ldots, t_{M-1})$ has one of the following two structures:

$$J_{M,L}(t_1, \ldots, t_{M-1}) = \sum_{k=1}^{M} \ell(t_{k-1}, t_k], \qquad 1 \le t_1 < t_2 < \ldots < t_{M-1} < L, \tag{3.1}$$

$$J_{M,L}(t_1, \ldots, t_{M-1}) = \prod_{k=1}^{M} \ell'(t_{k-1}, t_k], \qquad \ell'(t_{k-1}, t_k] \ge 0, \quad 1 \le t_1 < t_2 < \ldots < t_{M-1} < L. \tag{3.2}$$

where $\ell(p, q]$ and $\ell'(p, q]$ are called class cost (also called edge cost). A requirement is, that the class cost just depends on its borders, namely p and q (examples can be see in (5.3)(5.25)(5.27)(5.40)). In fact, only problems of the form (3.1) can be solved, but if the class costs $\ell'(p, q]$ are constrained to be positive, a problem of form (3.3) can be transformed into one of form (3.1), as shown below.

$$\begin{aligned} \arg\max \left\{ J'_{M,L}(t_1, \ldots, t_{M-1}) \right\} &= \arg\max \left\{ \prod_{k=1}^{M} \ell'(t_{k-1}, t_k] \right\} \\ = \arg\max \left\{ \log \left(\sum_{k=1}^{M} \ell'(t_{k-1}, t_k] \right) \right\} &= \arg\max \left\{ \sum_{k=1}^{M} \log \left(\ell'(t_{k-1}, t_k] \right) \right\} \\ = \arg\max \left\{ \sum_{k=1}^{M} \ell''(t_{k-1}, t_k] \right\}. & \end{aligned} \tag{3.3}$$

In the next few steps it is shown, that the thresholds for an objective function like (3.1) can be found using a DP algorithm. First a partial sum, up to gray level l for the first m

classes, is defined as

$$J_m(l) = \sum_{k=1}^{m} \ell(t_{k-1}, t_k] \,, \qquad 1 \leq t_1 < t_2 < \ldots < t_{m-1} < l. \tag{3.4}$$

For every gray level l, a subproblem can be defined as finding the optimal thresholds which partition the interval $[1, l]$ into m classes. The objective function of the subproblem is given by

$$J_m^*(l) = \max_{1 \leq t_1 < \ldots < t_{m-1} < l} \{J_m(l)\} \,. \tag{3.5}$$

By setting $m = M$ and $l = L$, this equation maximizes the overall problem and is equal to (3.1). By rewriting the objective function for the subproblems, the following recursion is obtained.

$$\begin{aligned} J_m^*(l) &= \max_{1 \leq t_1 < \ldots < t_{m-1} < l} \left\{ \sum_{k=1}^{m} \ell(t_{k-1}, t_k] \right\} \\ J_m^*(l) &= \max_{1 \leq t_1 < \ldots < t_{m-1} < l} \left\{ \sum_{k=1}^{m-1} \ell(t_{k-1}, t_k] + \ell(t_{m-1}, l] \right\} \\ J_m^*(l) &= \max_{1 \leq t_1 < \ldots < t_{m-1} < l} \left\{ J_{m-1}^*(t_{m-1}) + \ell(t_{m-1}, l] \right\} \end{aligned} \tag{3.6}$$

It is obvious to see, that if the thresholds of the subproblem $J_{m-1}^*(t_{m-1})$ are not set optimal and therefore are not maximizing the subproblem, that with these nonoptimal thresholds the objective function can never be maximal. This means, that only if the thresholds of the subproblems are set optimally, they can be part of the optimal solution of the total problem. For $m = 1$, the above sum of class costs cannot be taken apart anymore. Therefore, the recursive optimal cost with abort criterion is rewritten as

$$J_m^*(l) = \begin{cases} \max_{1 \leq t_1 < \ldots < t_{m-1} < l} \left\{ J_{m-1}^*(t_{m-1}) + \ell(t_{m-1}, l] \right\}, & \text{if } m > 1, \\ \ell(0, l], & \text{if } m = 1. \end{cases} \tag{3.7}$$

This equation states, that if the interval is not separated (just one class, $m = 1$) the optimal costs are the class costs $\ell(0, l]$, but if the interval is separated ($m > 1$), the optimal cost is the combination of optimal cost for $m - 1$ classes up to level t_{m-1} plus $\ell(0, l]$, which contributes the highest value. This idea can be visualized using a trellis structure, which is explained in the next section.

3.1 Trellis Structure

The trellis structure clarifies the task of finding the thresholds and gives a good understanding how the algorithm is implemented. The goal is to find the path connecting *start* and *end* which maximizes (3.7) for $m = M$ and $l = L$. The x-axis represents the gray levels l from $0 \ldots L$. Note that the zero value is just a placeholder for the start node. The y-axis shows the stage (m) of the algorithm. At stage m, the interval $[1, l]$ is divided into m classes by setting $m - 1$ thresholds. The gray dots are called nodes and are represented by $trellis(m, l)$. Because of the constraint that at stage m, $m - 1$ thresholds have to be set, and the thresholds can not be at the same position, the nodes exist just in a rhomboid. Every node in the trellis consists of two components, a value $trellis(m, l).L^*$ and a

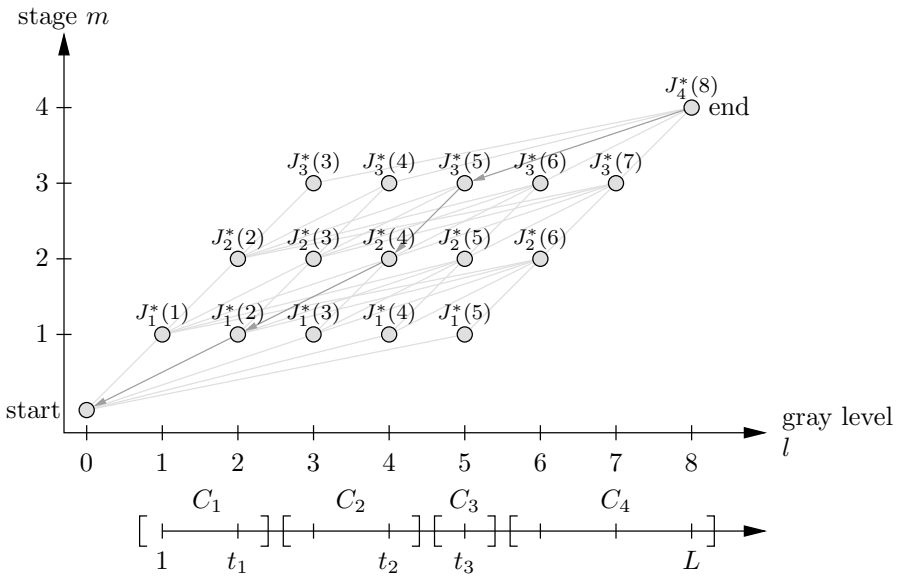

Figure 3.1: Trellis structure.

backpointer $trellis(m, l).pos^*$. The value stores the optimal partial sum up to this node, where the backpointer shows the position of the best node to come from.

The search for the best thresholds is processed as follows: At every node, the best node to come from and the resulting optimal cost is evaluated. The best path is stored in the node by setting the backpointer and setting the value of the node to the optimal cost so far. In the first stage ($m = 1$) the optimal cost at every node $trellis(1, l)$ is just the class cost $\ell(0, l]$ (see (3.7)) and the backpointer points to the start node. At every node in stages $1 < m < M$, the algorithm picks the path coming from nodes, which are one stage below and to the left of the current node, which contributes the highest cost. At the first node (leftmost) there is only one path possible, at the second node two paths have to be compared, at the third three and so on (possiple paths to each node are indicated as light gray lines). At stage $m = M$ only the optimal path to the end node has to be found. The optimal path through the structure and therefore the best set of thresholds can simply be found, by following the backpointers (backtracking) from end node, where the arrowheads indicate the positions of the thresholds.

The pseudocode in Algorithm 1 explains this process. The search is processed in four main parts. In the first step, the trellis is initialized at $stage = 1$. In the second step, the nodes in stages $1 < m < M$ are processed, and in the third step, the optimal path to the end node is evaluated. At the end, backtracking is used to find the thresholds.

For the search of the best path FINDOPTPATH(m, l) is called, which is explained in the pseudocode of Algorithm 2. For every node $trellis(m, l)$, FINDOPTPATH(m, l) checks every possible node to come from, and returns the optimal cost plus the best position to set the threshold, for this node.

3.2 Time Complexity

For the calculation of the timecomplexity, it is assumed that the sum $J_m^*(t_m) + \ell(t_m, l]$ can be calculated in $O(1)$ time. This is the case for all of the thresholding methods mentioned

Algorithm 1 DPSEARCH()

1: $---$ Stage 1
2: **for** $l \Leftarrow 1$ to $L - M + 1$ **do**
3: $\quad trellis(1, l).J^* \Leftarrow \ell(0, l]$
4: $\quad trellis(1, l).pos^* \Leftarrow 0$
5: **end for**
6: $---$ Stage $2 \ldots M - 1$
7: **for** $m \Leftarrow 2$ to $M - 1$ **do**
8: $\quad$ **for** $l \Leftarrow m$ to $L - M + m$ **do**
9: $\quad\quad (J_{max}, pos) \Leftarrow$ FINDOPTPATH(m, l)
10: $\quad\quad trellis(m, l).J^* \Leftarrow J_{max}$
11: $\quad\quad trellis(m, l).pos^* \Leftarrow pos$
12: $\quad$ **end for**
13: **end for**
14: $---$ Stage M
15: $(J_{max}, pos) \Leftarrow$ FINDOPTPATH(M, L)
16: $trellis(M, L).J^* \Leftarrow J_{max}$
17: $trellis(M, L).pos^* \Leftarrow pos$
18: $---$ Backtracking
19: $l \Leftarrow L$
20: **for** $m \Leftarrow M$ to 2 **do**
21: $\quad t_{m-1} = l \Leftarrow trellis(m, l).pos^*$
22: **end for**

Algorithm 2 FINDOPTPATH(m, l)

1: $J_{max} \Leftarrow -\infty$
2: **for** $i \Leftarrow m - 1$ to $l - 1$ **do**
3: $\quad J_{temp} \Leftarrow trellis(m - 1, i).J^* + \ell(i, l]$
4: $\quad$ **if** $J_{temp} > J_{max}$ **then**
5: $\quad\quad J_{max} \Leftarrow J_{temp}$
6: $\quad\quad pos \Leftarrow i$
7: $\quad$ **end if**
8: **end for**
9: **return** (J_{max}, pos)

in this thesis. Therefore, the time complexity of the DP algorithm is directly proportional to the number of times the above mentioned sum is calculated. Hence, the time complexity is $nr \cdot O(1)$, where nr is:

$$\begin{aligned} nr &= 2(L - M + 1) + (M - 2) \cdot \sum_{i=1}^{L-M+1} i \\ &= 2(L - M + 1) + (M - 2) \cdot \frac{1}{2}(L - M + 1)(L - M + 2) \\ &= 2M + \frac{7}{2}ML + \frac{1}{2}ML^2 - \frac{5}{2}M^2 - M^2L + \frac{1}{2}M^3 - L - L^2. \end{aligned} \tag{3.8}$$

Since it is assumed that $L >> M$, $\frac{1}{2}ML^2$ is the determining factor in (3.8) and the time complexity becomes $O(ML^2)$.

4 Improving the Dynamic Programming Approach

In the last chapter, a dynamic programming solution for the multilevel thresholding problem was shown. In this chapter, a method to reduce the time complexity of the dynamic programming algorithm is presented. Unlike the dynamic programming solution, which is applicable for many objective functions, the method introduced here can only be used if the objective function has certain properties. This kind of speedup for dynamic programming algorithms has been proposed for several problems ([3] [4] [5] [6] to cite only some which are relevant for our work) and is therefore not new. The main contribution of our work is the identification of a class of objective functions for which this method can be used. This class of objective functions is presented at the end of the chapter, after the improvement of the dynamic programming algorithm has been explained.

4.1 Definition of the Search Matrix

The DP algorithm employs a trellis structure which was explained in the last chapter. The algorithm proceeds from the bottom of the trellis to the top. For the nodes in the stages $2 \dots M-1$, the algorithm always compares paths emerging from nodes one stage lower and to the left of the current node. The problem of finding the optimal paths to all the nodes in one stage in the trellis is equivalent to the problem of finding the row wise maxima in a lower triangular matrix. This is ilustated in Figure 4.1.

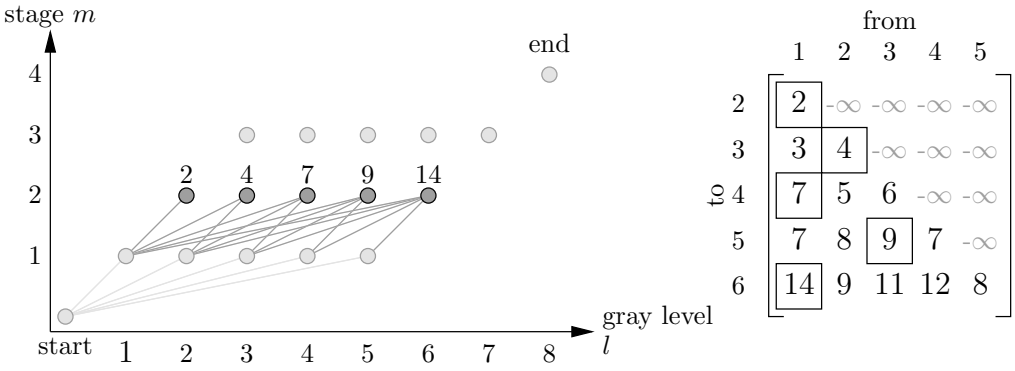

Figure 4.1: Equivalence to matrix search problem.

For the leftmost node, there is only one possible path, for the one right to it there are two and so on. In the search matrix, the cost of the paths up to all the nodes in one stage in the trellis are treated as matrix elements, where the column indicates where the path comes from and the row where the paths goes to, respectively. The elements in the upper triangular region of the matrix are defined to be $-\infty$, since there are no paths coming from nodes to the right or directly below the current node. The size of the search matrix

is $(L - M + 1) \times (L - M + 1)$, the matrix itself is defined as follows:

$$M(r,c) = \begin{cases} -\infty, & \text{if } c > r, \\ J^*_{m-1}(c+m-2) + \ell(c+m-2, r+m-1], & \text{if } c \leq r. \end{cases} \tag{4.1}$$

where m denotes the stage in the trellis and r and c the row and column index, respectively. It is obvious, that searching the row maxima in the lower triangular part of the matrix requires $O(L^2)$ time, this leads to the $O(ML^2)$ time complexity of the DP algorithm.

4.2 Quadrangle Inequality and Special Matrix Properties

It has already been shown, that the task of finding the optimal paths from the nodes in one class of the trellis to the nodes in next class is equal to finding the row wise maxima in a matrix. In this section, a property of the class cost $\ell(p,q]$ is introduced which leads to a search matrix with special properties. It will be shown in the next section, that the task of finding the row wise maxima in such a matrix is computionally less involved than finding the row wise maxima in a matrix without these properties. The property of the class cost which is introduced here is called *convex quadrangle inequality* (convex QI) and is defined as follows:

Definition 1. *The class cost $\ell(p,q]$ is said to fulfill the **convex quadrangle inequality** if the following is always true:*

$$\ell(a,u] + \ell(b,v] \geq \ell(a,v] + \ell(b,u], \quad 1 \leq a < b < u < v \leq L. \tag{4.2}$$

Figure 4.2 illustartes the intervals over which the class costs are calculated.

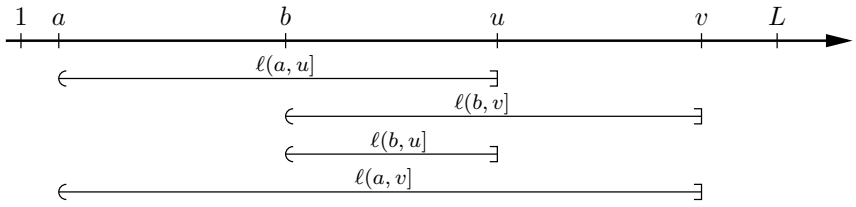

Figure 4.2: Class costs of overlapping intervals.

The previously defined search matrix at a stage m in the trellis, has a sum of the optimal cost up to a previous node and the class cost in each element. Now, take four elements from the lower triangular region of the matrix with rows $1 \leq r_1 < r_2 \leq L - M + 1$ and columns $1 \leq c_1 < c_2 \leq r_1$, build the sums of the top left and the lower right and the top right and the lower left element:

$$M(r_1,c_1) + M(r_2,c_2) \gtreqless M(r_1,c_2) + M(r_2,c_1). \tag{4.3}$$

since $J^*_{m-1}(c_1 + m - 2)$ and $J^*_{m-1}(c_2 + m - 2)$ are summands in both sums, they can be subtracted from both sides and the relation becomes:

$$\ell(c_1{+}m{-}2, r_1{+}m{-}1]{+}\ell(c_2{+}s{-}2, r_2{+}m{-}1] \gtreqless \ell(c_2{+}m{-}2, r_1{+}m{-}1]{+}\ell(c_1{+}m{-}2, r_2{+}m{-}1]. \tag{4.4}$$

The intervals over which the class costs are calculated are shown in Figure 4.3. If we know,

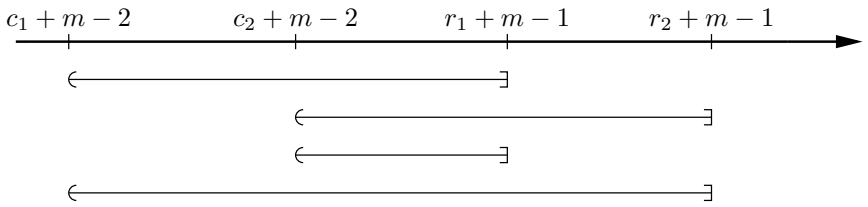

Figure 4.3: Intervals of the class costs in the matrix.

that $\ell(p, q]$ fulfills the convex QI this means:

$$M(r_1, c_1) + M(r_2, c_2) \geq M(r_1, c_2) + M(r_2, c_1). \tag{4.5}$$

A matrix which has this property is known as an inverse Monge matrix:

Definition 2. *The real $m \times n$ matrix M is called an **inverse Monge matrix** if M satisfies the **inverse Monge property**:*

$$M(i_1, k_1) + M(i_2, k_2) \geq M(i_1, k_2) + M(i_2, k_1), \quad 1 \leq i_1 < i_2 \leq m, \quad 1 \leq k_1 < k_2 \leq n. \tag{4.6}$$

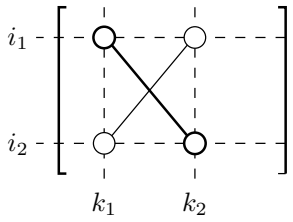

Figure 4.4: The inverse Monge property.

In Figure 4.4, an inverse Monge matrix is ilustrated. The the sum of the top left and the lower right element is always bigger than the sum of the lower left and the top right element. The elements in the upper triangular region of the search matrix were defined to be $-\infty$, since we want to find the row maxima. This definition is also needed for the search matrix to become inverse Monge. Monge matrices are named after the French engineer and mathematician Gaspard Monge (1746-l818) who discovered them. They arise in many optimization problems. An exstensive overview over Monge properties and their applications to optimization problems can be found in [7]. A property of inverse Monge matrices which is used extensively in this thesis, is the fact that inverse Monge matrices are always *totally monotone*:

Definition 3. *The real $m \times n$ matrix M is **totally monotone** if*

$$M(i_1, k_1) < M(i_1, k_2) \Longrightarrow M(i_2, k_1) < M(i_2, k_2), \quad 1 \leq i_1 < i_2 \leq m, \quad 1 \leq k_1 < k_2 \leq n. \tag{4.7}$$

Proof. Assume that matrix M is an inverse Monge matrix:

$$M(i_1, k_1) + M(i_2, k_2) \geq M(i_1, k_2) + M(i_2, k_1), \quad 1 \leq i_1 < i_2 \leq m, \quad 1 \leq k_1 < k_2 \leq n. \tag{4.8}$$

now assume $M(i_1, k_1) < M(i_1, k_2)$, because the matrix is Monge, we can make the following reasoning:

$$\begin{aligned} &M(i_1, k_1) < M(i_1, k_2) \\ &M(i_1, k_1) < M(i_1, k_2) \leq M(i_1, k_1) + M(i_2, k_2) - M(i_2, k_1) \\ &M(i_1, k_1) < M(i_1, k_1) + M(i_2, k_2) - M(i_2, k_1) \\ &M(i_2, k_1) < M(i_2, k_2) \end{aligned} \tag{4.9}$$

which means that the matrix is totally monotone. □

Totally monotone matrices are also monotone. Which means, that the row wise maxima in the matrix form a descending staircase.

Definition 4. *The real $m \times n$ matrix M is **monotone** if*

$$c_{\max}(i_1) \leq c_{\max}(i_2), \quad 1 \leq i_1 < i_2 \leq m. \tag{4.10}$$

where $c_{\max}(i)$ denotes the column index of the leftmost element conatining the maximum value of row i.

Proof. Assume the matrix M is totally monotone and $c_{\max}(i_1) > c_{\max}(i_2)$ for some $1 \leq i_1 < i_2 \leq m$, which means the matrix is not monotone. From the definition of a totally monotone matrix, we know:

$$M(i_1, c_{\max}(i_2)) < M(i_1, c_{\max}(i_1)) \Longrightarrow M(i_2, c_{\max}(i_2)) < M(i_2, c_{\max}(i_1)), \tag{4.11}$$

which contradicts the fact that $c_{\max}(i_2)$ is the position of maximum in row i_2. □

$$\begin{bmatrix} \mathbf{5} & 4 & 1 & 3 & 3 & 2 & 1 \\ 6 & \mathbf{8} & 2 & 1 & 0 & 3 & 2 \\ 5 & \mathbf{7} & 6 & 2 & 1 & 0 & 3 \\ 2 & 4 & \mathbf{5} & 2 & 2 & 1 & 0 \\ 7 & 1 & 0 & \mathbf{8} & 1 & 2 & 3 \\ 2 & 1 & 7 & 8 & \mathbf{9} & 1 & 2 \\ 5 & 2 & 1 & 7 & 8 & 1 & \mathbf{9} \end{bmatrix} \qquad \begin{bmatrix} \mathbf{5} & 2 & 1 & 2 & 3 & 1 & 2 \\ 2 & \mathbf{6} & 3 & 4 & 5 & 2 & 4 \\ 3 & 4 & 5 & 6 & \mathbf{7} & 1 & 6 \\ 2 & 3 & 4 & 5 & \mathbf{6} & 2 & 5 \\ 5 & 6 & 8 & 9 & 10 & 13 & \mathbf{15} \\ 3 & 5 & 7 & 11 & 15 & 16 & \mathbf{20} \\ 1 & 2 & 5 & 6 & 9 & 10 & \mathbf{11} \end{bmatrix}$$

Figure 4.5: A monotone and a totally monotone 7×7 matrix.

It is obvious, that knowledge about the monotonicity of the search matrix can be used to speedup the task of finding the row wise maxima. Two algorithms which solve this task efficiently are introduced in the next section.

4.3 Matrix Searching

Two efficient algorithms for finding the row wise maxima in monotone and totally monotone matrices are explained in this section. The first algorithm exploits only the monotonicity of the matrix, while the second algorithms requires a totally monotone matrix and

achieves an even lower time complexity. Both algorithms work with a implicitly defined matrix, which means a matrix entry is unknown until it is accessed by the algorithm. This is important, because the matrix searching algorithms are later used to reduce the time complexity of the dynamic programming algorithm. If the algorithm calculated every entry of the matrix, it would do the same amount of work as the normal search for the shortest path used in the dynamic programming algorithm and therefore not reduce the time complexity.

4.3.1 Divide-and-Conquer Algorithm for Monotone Matrices

The divide-and-conquer algorithm exploits the fact, that the row maxima in a monotone matrix build a staircase. First, it finds the maximum in the middle row of the matrix and is then executed recursively on two submatrices. The recursion stops when the matrix has only one row left. The pseudocode of Algorithm 3 explains the operation of the algorithm.

Algorithm 3 DIVCONQ(M)

```
1: [m, n] ⇐ size of M (rows, columns)
2: j ⇐ position leftmost maximum in row ⌈m/2⌉ of M
3: store the position of the maximum
4: if m = 1 then
5:    return
6: else
7:    if ⌈m/2⌉ ≠ 1 then
8:       A ⇐ submatrix with rows 1 to ⌈m/2⌉ − 1 and columns 1 to j of M
9:       DIVCONQ(A)
10:   end if
11:   B ⇐ submatrix with rows ⌈m/2⌉ + 1 to m and columns j to n of M
12:   DIVCONQ(B)
13: end if
```

4.3.1.1 Time Complexity

For the calculation of the time complexity, it is assumed that a matrix entry can be evaluated in $O(1)$ time. The time complexity of the algorithm is therefore directly proportional to the number of matrix entries that have to be evaluated until all the row maxima have been found. The algorithm is executed on a $m \times n$ matrix, for every recursion, the number of rows in the matrix is divided by two, which means that the maximal recursion depth is proportional to $\log_2(m)$. Searching the middle row for the maximum takes $O(n)$ time. In order to find the worst case time complexity, it is assumed that the maxima lie along the diagonal of the matrix. The time needed to find all the row maxima in the matrix, is expressed in (4.12).

$$T(m,n) = \begin{cases} O(n), & \text{if } m = 1, \\ 2T(m/2, n/2) + O(n), & \text{if } m > 1. \end{cases} \tag{4.12}$$

By introducing the constant time c needed to evaluate a matrix entry, this can be written

as:

$$T(m,n) = \begin{cases} cn, & \text{if } m = 1, \\ 2T(m/2, n/2) + cn, & \text{if } m > 1. \end{cases} \tag{4.13}$$

The solution to this recurrence can be found by using the recursion tree method [2], in Figure 4.6 the recursion tree for this algorithm is shown.

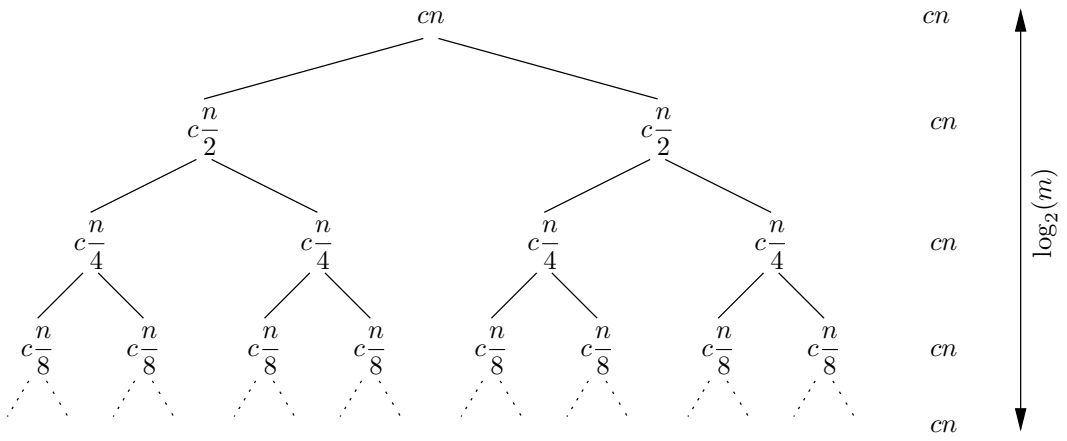

Figure 4.6: Recursion tree of the divide & conquer algorithm.

With help of the recursion tree, it is easy to see that the sum over all the nodes in a level is always cn. Since the tree has $\log_2(m)$ levels, the total sum becomes $cn \log_2(m)$, which means that the time complexity of the algorithm is $O(n \log m)$.

4.3.2 SMAWK Algorithm for Totally Monotone Matrices

The SMAWK algorithm [3] is named after its inventors Shor, Moran, Aggarwal, Wilbe and Klawe. Unlike the divide-and-conquer algorithm, the SMAWK algorithm does not work when the matrix is only monotone, it requires a totally monotone matrix. By exploiting not only the monotonicity, but the total monotonicity of the matrix, the SMAWK algorithm finds the row maxima of a $m \times n$ matrix ($m \leq n$) in $O(n)$ time, compared to $O(n \log m)$ time required by the divide-and-conquer algorithm. In this section, the functionality of the SMAWK algorithm is explained and the time complexity is derived. A more detailed explanation of the algorithm can be found in the original publication by Aggarwal et al. [3] and in [6].

Like the divide-and-conquer algorithm, the SMAWK algorithm searches the matrix recursively. The pseudocode in Algorithm 4 shows the structure of the algorithm. The core of the algorithm is the REDUCE function, which transforms the problem of finding the row wise maxima in an $m \times n$ ($m \leq n$) matrix, in the problem of finding the row wise maxima in a $m \times m$ matrix by deleting $n - m$ columns from the matrix. After the matrix has been reduced to an $m \times m$ matrix, the search algorithm is executed recursively on a matrix which contains only the even-numbered rows of the reduced matrix. The recursion stops, when REDUCE returns an 1×1 matrix, which is an element containing a row maximum. After that, the function MFILL finds the maxima in the odd-numbered rows of the matrix. Since the positions of the maxima in the even-numbered rows are already known from the recursive call of SMAWK, MFILL can find the maxima very efficiently.

The functions REDUCE and MFILL are explained next and the time complexities of these functions are analyzed. At the end of this section, the overall time complexity of the algorithm is derived.

Algorithm 4 SMAWK(M)

```
1: A ⇐ REDUCE(M)
2: if A is size 1 × 1 then
3:    store the position of A in M
4:    return
5: end if
6: B ⇐ matrix with only the even-numbered rows of A
7: SMAWK(B) {recursive call}
8: MFILL(A, B) {find the maxima in the odd rows of A}
```

As mentioned before, the REDUCE function plays a keyrole in the SMAWK algorithm. It deletes $n-m$ columns, which contain no row maxima, from the matrix. When REDUCE is executed on an $m \times n$ matrix, it can delete the columns in $O(n)$ time. The REDUCE function contains a case statement inside a while loop. The function returns when the matrix is square. In Algorithm 5, the structure of REDUCE is shown.

Algorithm 5 REDUCE(M)

```
1: A ⇐ M   k ⇐ 1
2: p ⇐ number of rows of A
3: while A has more columns than rows do
4:    case
5:       A(k, k) ≥ A(k, k + 1) and k < p :  {case a}
6:         k ⇐ k + 1
7:       A(k, k) ≥ A(k, k + 1) and k = p :  {case b}
8:         Delete column k + 1 of A
9:       A(k, k) < A(k, k + 1) :  {case c}
10:        Delete column k of A
11:        if k > 1 then
12:           k ⇐ k − 1
13:        end if
14:   end case
15: end while
```

Index k is used to access the matrix elements. Depending on the result of the comparison between $A(k, k)$ and $A(k, k + 1)$ and on the position in the matrix (index k) one of three possible branches is executed. If $A(k, k) \geq A(k, k + 1)$ and $k < p$ (branch a), the index k is simply increased, which means no maxima are in the elements of column $k + 1$, rows $1 \ldots k$. The second branch (branch b) is the same as the first branch but the algorithm compares elements in the last row of the matrix. Since no maxima can be in rows $1 \ldots m$ of column $k+1$, column $k+1$ can be deleted from the matrix. After deleting a column, the columns to the right of the deleted column are renumbered. In the third branch (branch c), column k is directly deleted and k is decreased. The proof, that the REDUCE function deletes only columns which contain no row maxima, is rather long and is not given here, it can be found in [3]. Since the renumbering of the columns makes it difficult to understand

the algorithm, the progress of the algorithm is illustrated in Figure 4.7. The algorithm compares elements which are shown in bold face, before an element is calculated the first time, it is shown in gray. Positions without a maximum have a gray background.

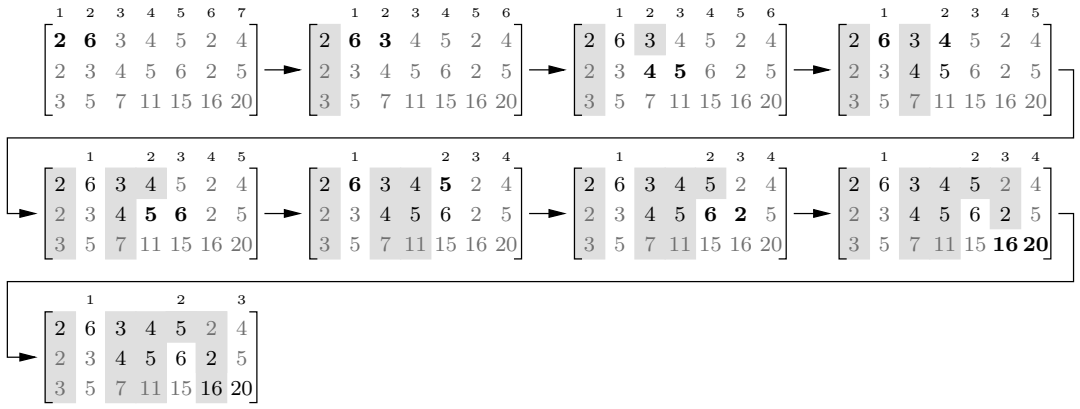

Figure 4.7: Operation of the REDUCE function.

It is shown in [3], that the REDUCE function reduces a $m \times n$ matrix to a $m \times m$ matrix in $O(n)$ time. For a better understanding, the proof is repeated here. The case statement in the while loop has three branches (a, b and c). Let the numbers a, b and c denote, respectively, the number of times the first, second and third branch is executed. Since columns are only deleted in the second and in the third branch and $n - m$ columns must be deleted, we know $b + c = n - m$. Furthermore, we know that the index k is only increased in the first branch and only decreased in the third branch. Since the index k always remains in the range $1 \dots m$, we know $a - c \leq m - 1$. The total number of times the while loop is executed can be denoted by t, which is $t = a + b + c$. Since every time the while loop is executed two matrix entries have to be evaluated, the time complexity of the algorithm is directly proportional to t. It is again assumed, that a matrix entry can be evaluated in $O(1)$ time. An upper bound for t is shown in (4.14).

$$t = a + b + c \leq n - m + a \leq n - m + m - 1 + c \leq 2n - m - 1. \tag{4.14}$$

Since $n \geq m$ and the evaluation of a matrix entry requires $O(1)$ time, this means REDUCE has a time complexity of $O(n)$.

After REDUCE returns a 1×1 matrix, the recursion stops and MFILL is executed. Since SMAWK has been recursively executed on a matrix with only the even-numbered rows, the positions of the maxima in the even-numbered rows are already known. The task of MFILL is to find the maxima in the odd-numbered rows. Since the maxima form a staircase in the matrix, REDUCE searches only the columns between the positions of the maxima in the row above and below the odd-numbered row. Algorithm 6 shows how MFILL finds the maxima in the odd-numbered rows.

The function MFILL always searches the maxima in matrix which has been reduced to a square matrix. The size of the matrix is therefore $m \times m$. Since MFILL only has to evaluate one matrix element for each column, the time complexity is $O(m)$.

Figure 4.8 shows the operation of the SMAWK algorithm. The initial call is on a 7×7 matrix, which means no columns are deleted by the REDUCE function. The first recursive call is on a 3×7 matrix with only the even numbered columns of the initial matrix. After REDUCE has deleted four columns from the matrix this matrix is becomes square. The

Algorithm 6 MFILL(A, B)

```
1: [m, n] ⇐ size of A (rows, columns)
2: mpos(2, 4, ··· , 2⌊m/2⌋) ⇐ positions of the maxima in the even-numbered rows of A
3: mpos(0) ⇐ 1   mpos(m + 1) ⇐ n
4: for i ⇐ 1 to ⌈m/2⌉ do
5:    row ⇐ 2i − 1
6:    max ⇐ −∞
7:    for col = mpos(row − 1) to mpos(row + 1) do
8:       if A(row, col) > max then
9:          max = A(row, col)
10:         mpos(row) = col
11:      end if
12:   end for
13: end for
```

second time SMAWK is executed recursively, the matrix has the size 1×3, REDUCE deletes two columns and the matrix becomes 1×1. At this point the recursion stops. Note, that the element of the last matrix is the maximum in the fourth row of the initial matrix. After the last recursive call of SMAWK returns, MFILL finds the maxima in the odd numbered rows of the 3×3 matrix. From the second recursive call of SMAWK, the position of the maximum in the row number two is already known (black border) and MFILL has only to search the elements which lie in a staircase (gray background). After MFILL has found all the maxima in the odd-numbered rows, the first recursive call of SMAWK returns. In the initial call, MFILL finds again the maxima in the odd-numbered rows, the elements which are searched are indicated by the gray background. After this, all the row wise maxima of the maxtrix have been found. Note, that the elements which are shaded gray have newer been evaluated during the search for the row wise maxima.

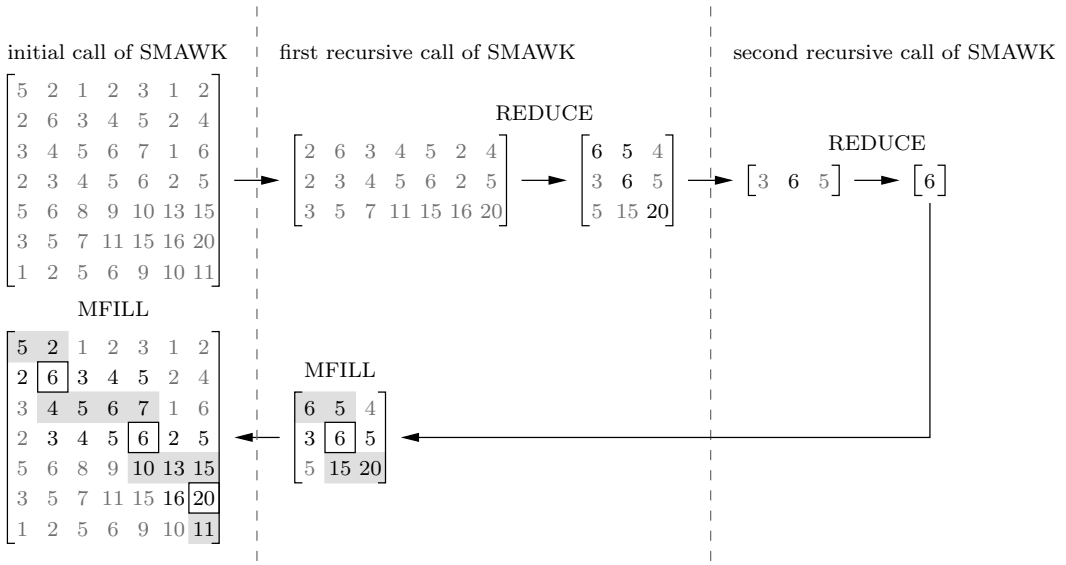

Figure 4.8: Operation of the SMAWK algorithm.

4.3.2.1 Time Complexity

The time complexities of the subroutines of the SMAWK algorithm have already been analyzed. When the algorithm is executed on a $m \times n$ ($m \leq n$) matrix, REDUCE requires $O(n)$ time and MFILL $O(m)$ time. Since the number of rows is always divided by two, the recursion depth is proportional to $\log_2(m)$. The overall time complexity is given by the recurrence in (4.15).

$$T(m,n) = \begin{cases} O(n), & \text{if } m = 1, \\ T(m/2, m) + O(m) + O(n), & \text{if } m > 1. \end{cases} \tag{4.15}$$

By assigning the time constants c_1, c_2 and c_3, this becomes

$$T(m,n) = \begin{cases} c_1 n, & \text{if } m = 1, \\ T(m/2, m) + c_2 m + c_3 n, & \text{if } m > 1. \end{cases} \tag{4.16}$$

The time $c_3 n$ only appears in the first call of the algorithm, in the recursive calls only the number of rows m of the initial matrix appears. The sum over the first call and all recursions therefore becomes

$$\begin{aligned} T(m,n) &= 2c_1 + c_2 m + c_3 n + \sum_{i=1}^{\lfloor \log_2(m) \rfloor} c_2 \frac{m}{2^i} + c_3 \frac{m}{2^{i-1}} \\ &= 2c_1 + c_2 m + c_3 n + \sum_{i=1}^{\lfloor \log_2(m) \rfloor} c_4 \frac{m}{2^i} \\ &< 2c_1 + c_2 m + c_3 n + c_5 m < c_6 m + c_3 n = O(n), \end{aligned} \tag{4.17}$$

which shows, that the algorithm has a time complexity of $O(n)$, since $n \geq m$. Unlike the divide-and-conquer, which calls itself two times and therefore creates a recursion tree in which each level has two times as many nodes as the level above (see Figure 4.6), the SMAWK algorithm calls itself only one time and only has one node per level of the recursion tree.

4.4 Combining DP and Matrix Searching

In order to reduce the time complexity of the dynamic programming algorithm introduced in Chapter 3, the matrix searching algorithms are combined with the DP algorithm. Like the normal DP algorithm, the algorithm first calculates the paths to the nodes in the first stage of the trellis, since there are $L - M + 1$ nodes and $L >> M$, this requires $O(L)$ time. After this, the matrix searching algorithm is executed at the stages $2 \ldots M - 1$, which means the matrix searching algorithm is executed $M - 2$ times. Every time the matrix searching algorithm needs the value of a matrix element, (4.1) is used to calculate its value. In other words, the matrix is defined implicitly and is never calculated and stored in the memory. The values of the row wise maxima found by the matrix searching algorithm are stored in the nodes of the stage where the matrix search is conducted. From the column indices of the maxima, the backpointers are set to point to the correct nodes in the stage below. Finding the optimal path to the end node in the last stage of the trellis, again requires $O(L)$ time.

Depending on the matrix searching algorithm used, either divide-and-conquer or SMAWK, different time complexities are achieved. Since the matrix has a size of $(L - M + 1) \times (L - M + 1)$, and the search is performed $M - 2$ times, the overall time complexity of the thresholding algorithm becomes $O(ML \log L)$ when the divide-and-conquer algorithm is used to find the maxima and $O(ML)$ when the SMAWK algorithm is used. Compared to the $O(ML^2)$ time complexity of the normal DP algorithm, the time complexities of these algorithms are significantly lower. As will be shown later, the reduced time complexity leads to shorter execution times and allows finding the optimal thresholds for pictures with more than 256 gray levels in reasonable time.

4.5 A Class of Objective Functions which fulfill the QI

Earlier in this chapter, it is shown that if the class cost $\ell(p, q]$ fulfill the convex quadrangle inequality, efficient algorithms can be employed to find the optimal thresholds. In this section, a generalized form of the class cost is presented, which always fulfills the convex quadrangle inequality and can be calculated in $O(1)$ time. The optimal thresholds, which maximize an objective function with class costs of this form, can therefore be found in $O(ML)$ time.

Theorem 1. *A class cost $\ell(p, q]$ of the form*

$$\ell(p, q] = w(p, q] \cdot f\left(\frac{\sum_{p<i\leq q} p(i) \cdot \gamma(i)}{w(p, q]}\right), \quad w(p, q] = \sum_{i=p+1}^{q} p(i), \tag{4.18}$$

where $w(p, q]$ is the class weight (probability of the class), $f(x)$ is a convex function on the interval $[\gamma(1), \gamma(L)]$ and the function $\gamma(x)$ is either monotone increasing or decreasing on the interval $[1, L]$, fulfills the convex quadrangle inequality.

The definition of a convex function and the proof of Theorem 1, are following.

Definition 5. *A function $f(x)$ is **convex** on an interval $[p, q]$ if for any two points x_1 and x_2 in $[p, q]$ and any λ where $0 < \lambda < 1$,*

$$f[\lambda x_1 + (1 - \lambda)x_2)] \leq \lambda f(x_1) + (1 - \lambda) f(x_2) \tag{4.19}$$

In Theorem 1 the statement is made, that the function $\gamma(x)$ can be either a monotone increasing or decreasing function. The following proof follows closely the one made to prove Theorem 3.6 in [8]. In our proof, it is assumed that $\gamma(x)$ is a monotone increasing function on the interval $[1, L]$. The proof for monotone decreasing functions is similar and not shown here.

Proof. Every monotone increasing function $\gamma(x)$ on the interval $[p, q]$ maps values $a < b < u < v$ to values $\gamma(a) < \gamma(b) < \gamma(u) < \gamma(v)$, as shown in Figure 4.10. The order of the elements remains the same.

The following expression, which is a part of (4.18), can be regarded as the mean calculated over the interval $(\gamma(p), \gamma(q)]$.

$$\frac{\sum_{p<i\leq q} p(i) \cdot \gamma(i)}{w(p, q]} = \mu_\gamma(q, p]. \tag{4.20}$$

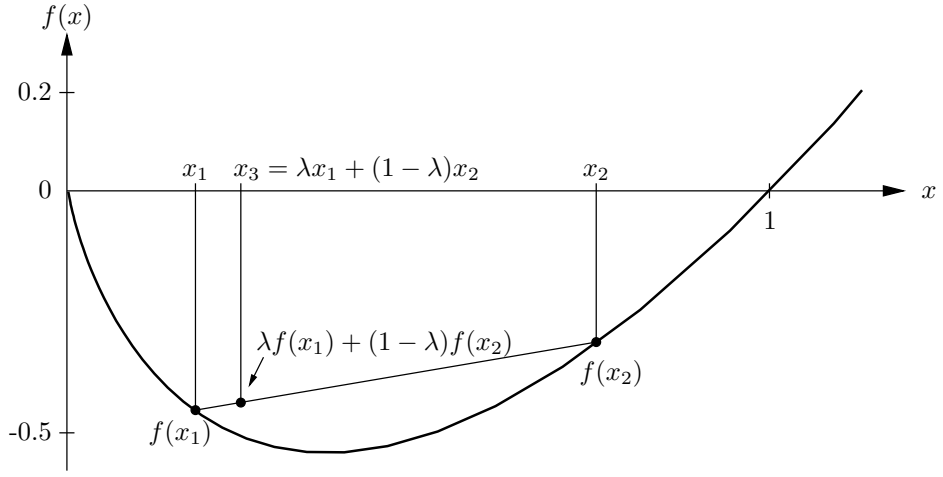

Figure 4.9: Example for a convex function, $f(x) = x \log_2(x)$ is convex on the interval $(0, \infty]$.

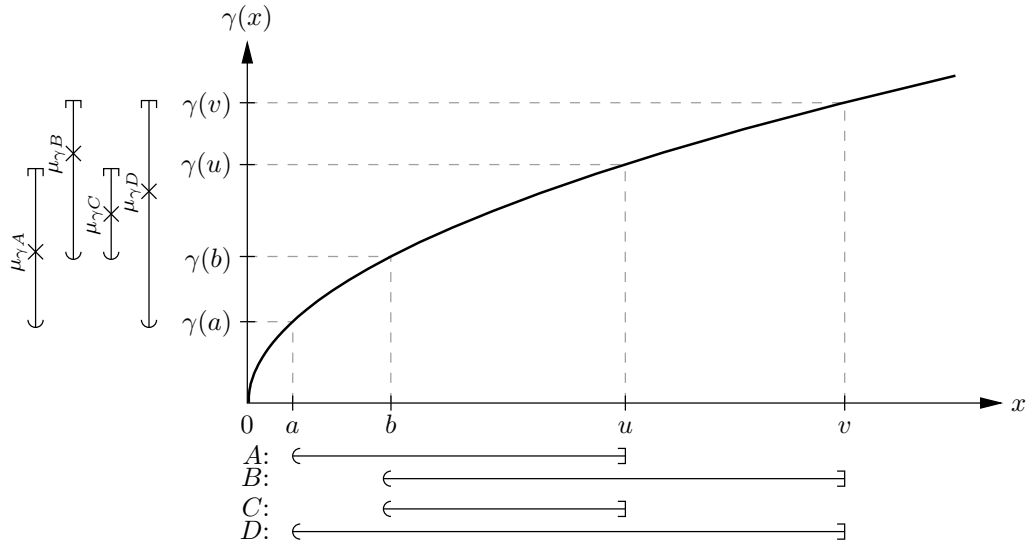

Figure 4.10: Mapping, $\gamma(x) = \sqrt{x}$ is monotone increasing on the interval $[0, \infty]$.

Since the mean $\mu(p, q]$ is monotone nondecreasing in p and q and we know that the order of the elements is not changed by the mapping, also $\mu_\gamma(q, p]$ is monotone nondecreasing in p and q. Therefore we have

$$\mu_{\gamma A} \leq \{\mu_{\gamma C}, \mu_{\gamma D}\} \leq \mu_{\gamma B}, \tag{4.21}$$

where A,B,C,D A are used to simplify the notation and are defined as shown in Figure 4.10. We can write $\mu_{\gamma C}$ and $\mu_{\gamma D}$ as linear combinations of $\mu_{\gamma A}$ and $\mu_{\gamma B}$, as

$$\mu_{\gamma C} = \alpha\mu_{\gamma A} + (1-\alpha)\mu_{\gamma B} \quad \Leftarrow \quad \alpha = \frac{\mu_{\gamma B} - \mu_{\gamma C}}{\mu_{\gamma B} - \mu_{\gamma A}}, \quad (1-\alpha) = \frac{\mu_{\gamma C} - \mu_{\gamma A}}{\mu_{\gamma B} - \mu_{\gamma A}}, \tag{4.22}$$

$$\mu_{\gamma D} = \beta\mu_{\gamma A} + (1-\beta)\mu_{\gamma B} \quad \Leftarrow \quad \beta = \frac{\mu_{\gamma B} - \mu_{\gamma D}}{\mu_{\gamma B} - \mu_{\gamma A}}, \quad (1-\beta) = \frac{\mu_{\gamma D} - \mu_{\gamma A}}{\mu_{\gamma B} - \mu_{\gamma A}}. \tag{4.23}$$

The goal is to show that $\ell(a,u] + \ell(b,v] \geq \ell(a,v] + \ell(b,u]$. Therefore, we want to show that

$$\ell_A + \ell_B - \ell_C - \ell_D \geq 0. \tag{4.24}$$

From (4.18), we obtain

$$\begin{aligned} \ell_A + \ell_B - \ell_C - \ell_D \; &= w_A \cdot f(\mu_{\gamma A}) + w_B \cdot f(\mu_{\gamma B}) \\ &\quad - w_C \cdot f(\mu_{\gamma C}) - w_D \cdot f(\mu_{\gamma D}). \end{aligned} \tag{4.25}$$

By replacing $f(\mu_{\gamma C})$ and $f(\mu_{\gamma D})$ by their upper bounds as shown in Figure 4.10, we have

$$\begin{aligned} \ell_A + \ell_B - \ell_C - \ell_D \; &\geq w_A \cdot f(\mu_{\gamma A}) + w_B \cdot f(\mu_{\gamma B}) \\ &\quad - w_C \cdot [\alpha f(\mu_{\gamma A}) + (1-\alpha) f(\mu_{\gamma B})] \\ &\quad - w_D \cdot [\beta f(\mu_{\gamma A}) + (1-\beta) f(\mu_{\gamma B})] \\ &= [w_A - \alpha w_C - \beta w_D] \cdot f(\mu_{\gamma A}) \\ &\quad + [w_B - (1-\alpha) w_C - (1-\beta) w_D] \cdot f(\mu_{\gamma B}). \end{aligned} \tag{4.26}$$

Below it is derived, that $[w_A - \alpha w_C - \beta w_D] = 0$.

$$\begin{aligned} & w_A - \alpha w_C - \beta w_D \\ &= w_A - \frac{\mu_{\gamma B} - \mu_{\gamma C}}{\mu_{\gamma B} - \mu_{\gamma A}} w_C - \frac{\mu_{\gamma C} - \mu_{\gamma A}}{\mu_{\gamma B} - \mu_{\gamma A}} w_D \\ &= \frac{1}{\mu_{\gamma B} - \mu_{\gamma A}} \Big[w_A(\mu_{\gamma B} - \mu_{\gamma A}) - w_C(\mu_{\gamma B} - \mu_{\gamma C}) - w_D(\mu_{\gamma B} - \mu_{\gamma D}) \Big] \\ &= \frac{1}{\mu_{\gamma B} - \mu_{\gamma A}} \left[w_A \left(\frac{N_B}{w_B} - \frac{N_A}{w_A} \right) - w_C \left(\frac{N_B}{w_B} - \frac{N_C}{w_C} \right) - w_D \left(\frac{N_B}{w_B} - \frac{N_D}{w_D} \right) \right] \\ &= \frac{1}{(\mu_{\gamma B} - \mu_{\gamma A}) w_B} \big[w_A N_B - w_B N_A - w_C N_B + w_B N_C - w_D N_B + w_B N_D \big], \end{aligned} \tag{4.27}$$

where $N(p,q] = \sum_{p<i\leq q} p(i) \cdot \gamma(i)$.

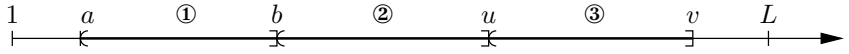

Figure 4.11: Intervals: ① $= (a,b]$, ② $= (b,u]$, ③ $= (u,v]$.

With writing $w(p,q]$ and $N(p,q]$ as partial sums over intervals shown in Figure 4.11, (4.27) becomes

$$\begin{aligned} & w_A - \alpha w_C - \beta w_D \\ &= \frac{1}{(\mu_{\gamma B} - \mu_{\gamma A}) w_B} \Big[(w_① + w_②)(N_② + N_③) - (w_② + w_③)(N_① + N_②) - w_②(N_② + N_③) \\ &\quad + (w_② + w_③) N_② - (w_① + w_② + w_③)(N_② + N_③) + (w_② + w_③)(N_① + N_② + N_③) \Big] \\ &= 0. \end{aligned} \tag{4.28}$$

In a similar way it can be shown that $[w_B - (1-\alpha) w_C - (1-\beta) w_D] = 0$. This proves Theorem 1. □

Theorem 2. *A class cost $\ell(p, q]$ of the form (4.18) can be calculated in $O(1)$ time after a preprocessing step which requires $O(L)$ time.*

Proof. The preprocessing step calculates two arrays, $W(i)$ and $N(i)$, they are defined recursively as

$$N(i) = \begin{cases} p(1) \cdot \gamma(1), & \text{if } i = 1, \\ N(i-1) + p(i) \cdot \gamma(i), & \text{if } 2 \leq i \leq L, \end{cases} \tag{4.29}$$

$$W(i) = \begin{cases} p(1), & \text{if } i = 1, \\ W(i-1) + p(i), & \text{if } 2 \leq i \leq L. \end{cases} \tag{4.30}$$

Since both arrays are L elements long, calculating and storing their values requires $O(L)$ time. After the arrays have been precalculated, the class cost $\ell(p, q]$ can be calculated as follows:

$$\ell(p, q] = [W(q) - W(p)] \cdot f\left(\frac{N(q) - N(p)}{W(q) - W(p)}\right). \tag{4.31}$$

When it is assumed that the time needed to calculate the value of the convex function $f(x)$ does not depend on x, as it is the case for most functions, (4.31) can be calculated by performing only lookup arithmetic operations. Therefore, the time needed to calculate $\ell(p, q]$ does not depend on the values of p and q, which proves that it can be calculated in $O(1)$ time. □

5 Efficient Algorithms For Known Thresholding Methods

So far, algorithms based on dynamic programming and matrix searching for multilevel thresholding have been introduced. In addition, a class of objective functions, for which the optimal thresholds can be found in $O(ML)$ time, has been identified. However, specific objective functions, which have been proposed in the literature for multilevel thresholding, have not yet been discussed. In this chapter, four different thresholding methods and their objective functions are reviewed. The optimal thresholds for all these methods can be found by the dynamic programming algorithm with $O(ML^2)$ time complexity. For some methods, also the faster algorithms, which combine dynamic programming and matrix searching, can be employed.

The knowledge, that dynamic programming can be used to find the optimal thresholds, is not new for all methods shown. Our contribution in this chapter is, that we propose the use of the dynamic programming algorithm for maximum entropy thresholding [9]. We also show, that the optimal thresholds for the method proposed by N. Otsu [10] can be found in $O(ML)$ time. Finally, we extend the minimum cross entropy method [11] to multiple thresholds and propose the use of an algorithm which finds the optimal thresholds in $O(ML)$ time.

5.1 Maximum Entropy Thresholding

Maximum entropy thresholding refers to a class of thresholding methods which try to maximize the sum of the entropies of the classes and therefore their information content. A well known maximum entropy method is the one proposed by Kapur et al. [9]. For multiple classes, the optimal thresholds are found by maximizing the following objective function:

$$J_{M,L}(t_1,\ldots,t_{M-1}) = \sum_{k=1}^{M} \sum_{i=t_{k-1}+1}^{t_k} \frac{p(i)}{w(t_{k-1},t_k]} \log\left(\frac{p(i)}{w(t_{k-1},t_k]}\right). \tag{5.1}$$

Obviously, an exhaustive search can be used to find the optimal thresholds. However, due to the high time complexity this is not desireable. Several iterative methods have been proposed which find the thresholds faster. In [12] using an iterative algorithm based on ICM (iterated conditional modes) is proposed. The proposed algorithm has a time complexity of $O(ML^2)$. A problem with iterative algorithms is, that they are not always guaranteed to find the optimal thresholds. Also, the exact number of iterations needed until good thresholds are found and therefore the execution time of the algorithm depends on the structure of the histogram. We propose, that the optimal thresholds which maximize (5.1), can be found in $O(ML^2)$ time by using the dynamic programming algorithm

introduced in Chapter 3. For this, (5.1) is rewritten with class costs:

$$J_{M,L}(t_1, \ldots, t_{M-1}) = \sum_{k=1}^{M} \ell(t_{k-1}, t_k]. \tag{5.2}$$

Where the class cost $\ell(p, q]$ is defined as

$$\ell(p, q] = -\sum_{i=p+1}^{q} \frac{p(i)}{w(p, q]} \log\left(\frac{p(i)}{w(p, q]}\right). \tag{5.3}$$

Note, the cost of class C_k depends only on its borders, which means on t_{k-1} and t_k. Therefore, the dynamic programming algorithm can be employed for finding the optimal thresholds. The time complexity of the dynamic programming algorithm only is $O(ML^2)$, if the class cost $\ell(p, q]$ can be computed in $O(1)$ time. This is possible by introducing a preprocessing step similar to the one explained in Section 4.5.

For the preprocessing step, (5.3) is rewritten as

$$\begin{aligned}
\ell(p, q] &= \sum_{i=p+1}^{q} \frac{p(i)}{w(p, q]} \log(w(p, q]) - \sum_{i=p+1}^{q} \frac{p(i)}{w(p, q]} \log(p(i)) \\
&= \frac{\log(w(p, q])}{w(p, q]} \cdot \sum_{i=p+1}^{q} p(i) - \frac{1}{w(p, q]} \cdot \sum_{i=p+1}^{q} p(i) \cdot \log(p(i)) \\
&= \log(w(p, q]) - \frac{1}{w(p, q]} \cdot \sum_{i=p+1}^{q} p(i) \cdot \log(p(i)).
\end{aligned} \tag{5.4}$$

Two arrays, both with length L, can be calculated in $O(L)$ time:

$$H(i) = \begin{cases} p(1) \cdot \log(p(1)), & \text{if } i = 1, \\ H(i-1) + p(i) \cdot \log(p(i)), & \text{if } 2 \leq i \leq L. \end{cases} \tag{5.5}$$

$$W(i) = \begin{cases} p(1), & \text{if } i = 1, \\ W(i-1) + p(i), & \text{if } 2 \leq i \leq L. \end{cases} \tag{5.6}$$

After the all values of $N(i)$ and $W(i)$ have been precalculated, the class cost $\ell(p, q]$ can be calculated in $O(1)$ time:

$$\ell(p, q] = \log(W(q) - W(p)) - \frac{H(q) - H(p)}{W(q) - W(p)}. \tag{5.7}$$

Therefore, it is possible to find the optimal thresholds in $O(ML^2)$ time. If the dynamic programming algorithm or the algorithm proposed in [12] finds the thresholds faster is not clear. However, our algorithm has the same time complexity and is guaranteed to find the optimal thresholds.

5.2 Otsu's Thresholding Criterion

Because of simplicity and robustness the method proposed in 1979 by N. Otsu [10] is widely used and referenced in numerous papers on image thresholding. In the original

paper, the problem is first shown for two classes (one threshold) and later extended to a problem with multiple thresholds. The chosen notation is similar to the one Otsu used, but adapted to better suite the multilevel thresholding case.

For the two class case the optimal threshold according to Otsu, is the threshold, which minimizes the sum of the weighted class variances. Otsu calls this sum within-class variance, and defines it as

$$\sigma_W^2 = w_1\sigma_1^2 + w_2\sigma_2^2. \tag{5.8}$$

The criterion tries to separate the pixels, such that the classes are homogeneous in themselves. Since a measure of group homogeneity is the variance, the Otsu criterion follows consequently. Therefore, the optimal threshold is the one, for which the within-class variance is minimal.

In order to find the optimal threshold, instead of minimizing the within-class variance, the between-class variance can be maximized. The between class variance is defined as follows:

$$\sigma_B^2 = w_1(\mu_1 - \mu_T)^2 + w_1(\mu_2 - \mu_T)^2\,, \qquad \mu_T = \sum_{i=1}^{L} p(i)\cdot i, \tag{5.9}$$

where μ_T is the total mean calculated over all gray levels. This follows from the fact, that the sum of the within-class variance and the between-class variance is equal to the total variance σ_T^2, which is independent of the threshold and therefore constant.

$$\sigma_W^2 + \sigma_B^2 \;=\; \sigma_T^2 \;=\; constant\,, \quad \sigma_T^2 = \sum_{i=1}^{L} p(i)\cdot(i-\mu_T)^2. \tag{5.10}$$

Proof. The variance can be rewritten as

$$\begin{aligned}
\sigma_T^2 &= \sum_{i=1}^{t} p(i)\cdot(i-\mu_1+\mu_1-\mu_T)^2 + \sum_{i=t+1}^{L} p(i)\cdot(i-\mu_2+\mu_2-\mu_T)^2 \\
&= \sum_{i=1}^{t} p(i)\cdot\left[(i-\mu_1)^2 + 2(i-\mu_1)(\mu_1-\mu_T) + (\mu_1-\mu_T)^2\right] \\
&\quad + \sum_{i=t+1}^{L} p(i)\cdot\left[(i-\mu_2)^2 + 2(i-\mu_2)(\mu_2-\mu_T) + (\mu_2-\mu_T)^2\right].
\end{aligned} \tag{5.11}$$

Since

$$\begin{aligned}
\sum_{i=1}^{t} p(i)\cdot(i-\mu_1) \quad &= \quad \sum_{i=1}^{t} p(i)\cdot i - \mu_1\cdot\sum_{i=1}^{t} p(i) \;=\; 0, \\
&\Rightarrow \quad \sum_{i=1}^{t} p(i)\cdot 2(i-\mu_1)(\mu_1-\mu_T) \;=\; 0, \\
&\Rightarrow \quad \sum_{i=t+1}^{L} p(i)\cdot 2(i-\mu_2)(\mu_2-\mu_T) \;=\; 0,
\end{aligned} \tag{5.12}$$

we can rewrite σ_T^2 as

$$\begin{aligned}\sigma_T^2 =& \sum_{i=1}^{t} p(i)\cdot(i-\mu_1)^2 \;+\; w_1(\mu_1-\mu_T)^2 \\ &+ \sum_{i=t+1}^{L} p(i)\cdot(i-\mu_2)^2 \;+\; w_2(\mu_2-\mu_T)^2 \\ =& \Big[w_1\sigma_1^2 + w_2\sigma_2^2\Big] \;+\; \Big[w_1(\mu_1-\mu_T)^2 + w_2(\mu_2-\mu_T)^2\Big] \\ =& \sigma_W^2 + \sigma_B^2. \end{aligned} \tag{5.13}$$

□

Extended to multilevel thresholding, the within-class variance and the between-class variance can be written as follows.
within-class variance:

$$\sigma_W^2 = \sum_{k=1}^{M} w_k\sigma_k^2. \tag{5.14}$$

between-class variance:

$$\sigma_B^2 = \sum_{k=1}^{M} w_k(\mu_k-\mu_T)^2. \tag{5.15}$$

The equation (5.10) still holds for more than one threshold. So the task of finding the optimal set of thresholds $[t_1^*, t_2^*, \ldots, t_{M-1}^*]$ is either to find the thresholds, which minimize the within-class variance, or to find the ones, which maximize the between-class variance. The result is the same.

$$[t_1^*, t_2^*, \ldots, t_{M-1}^*] \;=\; \arg\min\{\sigma_W^2\} \;=\; \arg\max\{\sigma_B^2\}. \tag{5.16}$$

In this case, σ_W^2 and σ_B^2 represent two different objective functions as defined in Section 2.1. If the within-class variance is rewritten with the interval notation as introduced in Chapters 2, we have

$$\sigma_W^2 = \sum_{k=1}^{M} w(t_{k-1}, t_k]\sigma^2(t_{k-1}, t_k]\,, \tag{5.17}$$

It is easy to see, that the within-class variance defined by Otsu, has the structure defined in (3.1). Therefore, the DP algorithm can be employed to find the optimal thresholds, as proposed by N. Otsu in [13].

A problem equivalent to finding the optimal threshold for the Otsu criterion, as written in (5.17), is encountered in optimal scalar quantizer design. A scalar quantizer, partitions the dynamic range of an input signal into K intervals, where a representative is assigned to each interval. An optimal scalar quantizer, as defined by J. Max [14], minimizes the expected mean square quantization error. The amplitude density function of a digital input signal, can be represented by a histogram of N points. An optimal scalar quantizer minimizes the following objective function:

$$E(\mathbf{q}) = \sum_{j=1}^{K} \sum_{i=q_{j-1}+1}^{q_j} P(x_i)\cdot(x_i-r_j)^2. \tag{5.18}$$

where r_j is the representative of interval j. For minimal mean square quantization error, the representative r_j has to be the mean of the corresponding interval:

$$r_j = \mu(q_{j-1}, q_j]. \tag{5.19}$$

Therefore, an optimal scalar quantizer minimizes $E(\mathbf{q})$ subject to

$$1 \leq q_1 < q_2 < \ldots < q_{K-1} < M. \tag{5.20}$$

Note, that this is exactly the same, as finding the optimal thresholds for the Otsu criterion, where $x_i = i$. This is easy to see, if the variance $\sigma^2(t_{k-1}, t_k]$ in (5.17) is replaced by its definition.

$$\sigma_W^2 = \sum_{k=1}^{M} w(t_{k-1}, t_k] \frac{\sum_{t_{k-1}<i\leq t_k} p(i) \cdot (i - \mu(t_{k-1}, t_k])^2}{w(t_{k-1}, t_k]}, \tag{5.21}$$

$$1 \leq t_1 < t_2 < \ldots < t_{M-1} < L.$$

For optimal scalar quantization, X. Wu showed in [4] and [5], that the optimal quantizer $\mathbf{q}$ can be found in $O(KN \log N)$ and $O(KN)$ time, by using the algorithms presented in Chapter 4. Therefore, also for the Otsu criterion the thresholds can be found in $O(ML \log L)$ and $O(ML)$ time, respectively.

This statement can also be drawn, by showing that the objective function for the Otsu criterion has the structure shown in Theorem 1.

In [15] it has been shown, that the between-class variance criterion can be modified, such that the objective function can be calculated more efficiently. The modified between-class variance can be written as

$$\begin{aligned}
\sigma_B^2 &= \sum_{k=1}^{M} w_k(\mu_k - \mu_T)^2 \\
&= \sum_{k=1}^{M} w_k(\mu_k^2 - 2\mu_k\mu_T + \mu_T^2) \\
&= \sum_{k=1}^{M} w_k\mu_k^2 \; - \; 2\mu_T \sum_{k=1}^{M} w_k\mu_k \; + \; \mu_T^2 \sum_{k=1}^{M} w_k \\
&= \sum_{k=1}^{M} w_k\mu_k^2 \; - \; \mu_T^2,
\end{aligned} \tag{5.22}$$

because

$$\sum_{k=1}^{M} w_k\mu_k = \mu_T \quad \text{and} \quad \sum_{k=1}^{M} w_k = 1. \tag{5.23}$$

For the search of the optimal thresholds, the total variance μ_T^2 in (5.22) can be omitted. Therefore, the objective function for the modified Otsu criterion is defined as

$$J_{M,L}(t_1, \ldots, t_{M-1}) = \sum_{k=1}^{M} w(t_{k-1}, t_k] \cdot (\mu(t_{k-1}, t_k])^2, \tag{5.24}$$

Therefore, the class cost $\ell(p, q]$ are given by

$$\ell(p, q] = w(p, q] \cdot (\mu(p, q])^2. \tag{5.25}$$

Since, $f(x) = x^2$ is convex, and $\gamma(i) = i$ is monotone increasing, the class cost of the Otsu criterion always fulfills the convex quadrangle inequality, as shown in Section 4.5. Therefore, the optimal thresholds can be found in $O(ML \log L)$ and $O(ML)$ time.

It is interesting, that even though N. Otsu proposed a dynamic programming algorithm with a time complexity $O(ML^2)$ [13] for his method and the connection to scalar quantization has been realized [16], no optimal algorithms with lower time complexities have been proposed so far.

5.3 Kittler and Illingworth's Thresholding Criterion

The Kittler and Illingworth thresholding method [17], assumes that the populations in the histogram are distributed normally, with distinct means and variances. The proposed method optimizes a criterion related to the average pixel classification error rate [18]. The criterion for multilevel thresholding, which has to be minimized is given as

$$J(t_1, \ldots, t_{M-1}) = \sum_{k=1}^{M} w_k \cdot \log\left(\frac{\sigma_k}{w_k}\right). \tag{5.26}$$

Written with the interval notation, the class cost $\ell(p, q]$ for this criterion is consequently given by

$$\ell(p, q] = w(p, q] \cdot \log\left(\frac{\sigma(p, q]}{w(p, q]}\right). \tag{5.27}$$

The objective function has obviously the form shown in (3.1). Therefore, the DP algorithm presented in Chapter 3 can be employed to find the optimal thresholds in $O(ML^2)$ time, as shown in [18].

The criterion shown in (5.26) reflects indirectly the overlap between the Gaussian models as shown in Figure 5.1. Every class of pixels is represented as a Gaussian model with the mean μ_k and the variance σ_k. The optimal thresholds are the ones which minimize the overlap between these models.

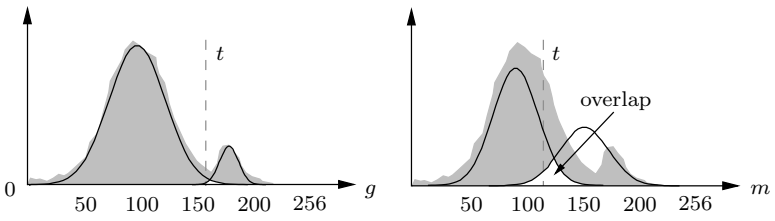

Figure 5.1: Simple example with two classes, for a good and a bad threshold.

5.4 Minimum Cross Entropy Thresholding

The idea behind the minimum cross entropy method proposed by C. Li and C. Lee [11] is to minimize the cross entropy between the image and the segmented version. The method has only been proposed for one threshold, but the extension to multiple thresholds is straightforward, as will be shown later. First, the method is explained for one threshold, as in the original paper, and then extended to multiple thresholds.

The optimal threshold for the minimum cross entropy method minimizes the following objective function:

$$\eta(t) = \sum_{f_j<t} f_i \log\left(\frac{f_j}{r_1}\right) + \sum_{f_j\geq t} f_j \log\left(\frac{f_j}{r_2}\right), \tag{5.28}$$

where f_j is the gray level value of pixel j and r_n is its representative in the binarized image. Note, that the objective function is calculated over the whole image, not only over the histogram. The objective function 5.28 can be seen as the Kullback-Leibler (KL) distance between two distributions. The first distribution contains the gray level values of the pixels in the image and the second distribution the gray level values of the pixels of the binarized image. The number of values of both distributions is the same as the number of pixels in the image. In order to have a binarized image which has the same total intensity as the original image, the reconstruction values are restricted by the following constraints:

$$\sum_{f_j<t} f_j = \sum_{f_j<t} r_1, \tag{5.29}$$

$$\sum_{f_j\geq t} f_i = \sum_{f_j<t} r_2. \tag{5.30}$$

The constraints can be met, if the reconstruction values are set to the mean gray level value of the corresponding class:

$$r_1 = \mu_1, \tag{5.31}$$

$$r_2 = \mu_2. \tag{5.32}$$

In (5.28), the pixel with gray level value f_j is encountered $h(f_j)$ times in the sum, where $h(f_j)$ denotes histogram value for a pixel with the gray level value f_j. Therefore, the redundant summations can be grouped together by employing the histogram of the image:

$$\eta(t) = \sum_{i=1}^{i=t-1} h(i)\cdot i \cdot \log\left(\frac{i}{\mu_1(t)}\right) + \sum_{i=t}^{i=L} h(i)\cdot i \cdot \log\left(\frac{i}{\mu_2(t)}\right). \tag{5.33}$$

In the notation used in [11], the first class contains the pixels with gray level value $1 \ldots t-1$ and the second class the pixels with the gray level values $t \ldots L$. For the rest of this section, this definition is modified to the one used throughout the rest of this thesis, as defined in Chapter 2.

The extension of (5.33) to more than one threshold is straightforward:

$$\eta_{M,L}(t_1,\ldots,t_{M-1}) = \sum_{k=1}^{M} \sum_{i=t_{k-1}+1}^{t_k} h(i)\cdot i \cdot \log\left(\frac{i}{\mu(t_{k-1},t_k]}\right). \tag{5.34}$$

The optimal thresholds are found by minimizing (5.34):

$$[t_1^*, t_2^*, \ldots, t_{M-1}^*] = \underset{0<t_1<\ldots<t_{M-1}<L}{\arg\min} \left\{\eta_{M,L}(t_1,\ldots,t_{M-1})\right\}. \tag{5.35}$$

This objective function can be further simplified:

$$\eta_{M,L}(t_1,\ldots,t_{M-1}) = \sum_{k=1}^{M} \sum_{i=t_{k-1}+1}^{t_k} h(i)\cdot i \cdot \log(i) \; - \sum_{k=1}^{M} \sum_{i=t_{k-1}+1}^{t_k} h(i)\cdot i \cdot \log\left(\mu(t_{k-1},t_k]\right). \tag{5.36}$$

Note, that the value of the first sum does not depend on the positions of the thresholds. Therefore, the first sum can be left out of the calculation. By not calculating the first sum and using the normalized histogram $p(i)$ instead of the histogram $h(i)$ a new objective function is found:

$$J_{M,L}(t_1,\ldots,t_{M-1}) = \sum_{k=1}^{M} \sum_{i=t_{k-1}+1}^{t_k} p(i)\cdot i \cdot \log\big(\mu(t_{k-1},t_k]\big). \tag{5.37}$$

Maximizing this objective function results in the same thresholds as minimizing (5.34):

$$\operatorname*{arg\,min}_{0<t_1<\ldots<t_{M-1}<L} \{\eta_{M,L}(t_1,\ldots,t_{M-1})\} = \operatorname*{arg\,max}_{0<t_1<\ldots<t_{M-1}<L} \{J_{M,L}(t_1,\ldots,t_{M-1})\}. \tag{5.38}$$

Obviously, the objective function (5.37) can be written with class costs, where the class cost is defined as:

$$\ell(p,q] = \sum_{i=p+1}^{q} p(i)\cdot i \cdot \log\big(\mu(p,q]\big). \tag{5.39}$$

The factor $\log(\mu(p,q])$ is constant for fixed p and q. Therefore, it can be taken out of the sum:

$$\ell(p,q] = \log\big(\mu(p,q]\big) \cdot \sum_{i=p+1}^{q} p(i)\cdot i. \tag{5.40}$$

Note, that $\sum_{i=p+1}^{q} p(i)\cdot i = w(p,q]\cdot\mu(p,q]$, where $w(p,q]$ is the weight of the class. By using this, the class cost can be written as

$$\ell(p,q] = w(p,q]\cdot\mu(p,q]\cdot\log\big(\mu(p,q]\big). \tag{5.41}$$

Note, that $\mu(p,q]$ is the mean calculated over the interval $(p,q]$, which is defined as

$$\mu(p,q] = \frac{\sum_{i=p+1]}^{q} p(i)\cdot i}{w(p,q]}. \tag{5.42}$$

Since the function $f(x) = x\log(x)$ is convex on the interval $(0,\infty]$ and $\gamma(i) = i$ is monotone increasing, the class cost has the form defined in (4.18). This means, the class cost fulfills the convex quadrangle inequality and it can be calculated in $O(1)$ time after a preprocessing step. Therefore, the optimal thresholds for the method proposed in [11], which has been extended to multiple thresholds in this thesis, can be found in $O(ML)$ time by using the algorithm combining dynamic programming and SMAWK matrix searching.

6 Implementations for the Otsu criterion

In order to find out how the time complexities of the algorithms affect their execution time, the thresholding algorithms introduced in this thesis are implemented. Since it is one of the most prominent thresholding methods and its class cost fulfills the convex quadrangle inequality, the objective function of the Otsu method is used for the implementations. All algorithms return the same optimal thresholds, therefore only the execution time and the memory required can be used for a performance comparison. Consequently, the implementation of the algorithms must be as efficient as possible. This is achieved by using ANSI C for the implementations and allowing no dynamic memory allocations during the execution of the algorithms. Of course, it would be possible to further reduce the execution times of the algorithms by implementing them using assembly. But since the algorithms are rather complex and the overhead incurred by using ANSI C is about the same for all implementations, this has not been attempted. The divide-and-conquer and the SMAWK algorithms are recursive. A problem with recursive algorithms is, that a lot of memory is needed to pass the function arguments and save the return addresses, if more memory is needed than available, a stack overflow occurs. This is avoided by using global variables whenever possible. Like this, the functions have fewer arguments and therefore require less memory on the stack. A drawback of using global variables is, that the implementations are not thread safe, which means they cannot be used by concurrent threads. Since the code of the implementations is quite long, it is not included in the thesis. In this chapter only the important concepts of the implementations are explained. For a full reference, the actual ANSI C code, which is availale on the internet and on the CD, should be consulted. The notation used in this chapter is the same as used in the rest of this thesis. In order to avoid confusion, the gray levels of an image are still defined to go from 1 to L, even though for the actual implementations 0 to $L-1$ is used, as this corresponds directly to the values of the pixels in a gray scale image.

As shown before, the optimal thresholds for the Otsu method can be found by maximizing the following objective function:

$$J_{M,L}(t_1,\ldots,t_{M-1}) = \sum_{k=1}^{M} w(t_{k-1},t_k] \cdot (\mu(t_{k-1},t_k])^2. \tag{6.1}$$

For the calculation of the time complexities, it was always assumed that the class cost can be calculated in $O(1)$ time. This can be achieved by performing a preprocessing step. The preprocessing step is the same as shown in Section 4.5, for completeness it is repeated here and applied to the Otsu method. The preprocessing step is the same for all implementations. By further simplifying (6.1), it becomes:

$$J_{M,L}(t_1,\ldots,t_{M-1}) = \sum_{k=1}^{M} \frac{(\sum_{i=t_{k-1}+1}^{t_k} p(i)\cdot i)^2}{w(t_{k-1},t_k]}. \tag{6.2}$$

Now, two arrays, $N(i)$ and $W(i)$ are introduced. Both are L elements long and are defined

as follows:

$$N(i) = \begin{cases} p(1), & \text{if } i = 1, \\ N(i-1) + p(i) \cdot i, & \text{if } 2 \leq i \leq L. \end{cases} \tag{6.3}$$

$$W(i) = \begin{cases} p(1), & \text{if } i = 1, \\ W(i-1) + p(i), & \text{if } 2 \leq i \leq L. \end{cases} \tag{6.4}$$

Obviously, filling in the values of $N(i)$ and $W(i)$ can be done in $O(L)$ time. After this, the class cost $\ell(p, q]$ can be calculated in $O(1)$ time by performing some lookup and arithmetic operations:

$$\ell(p, q] = \frac{(N(q) - N(p))^2}{W(q) - W(p)}. \tag{6.5}$$

For the case $W(q) - W(p) = 0$, which means the probability of the class is zero and a division zero would occur if the value was calculated directly, $\ell(p, q]$ is set to zero. This preprocessing step is essentially the same as the one advocated in [15], although in [15] the authors go one step further and build a lookup table for every possible combination of p and q ($0 \leq p < q \leq L$). Using this lookup table can increase the speed of an exhaustive search, as shown in [15], but is not desireable in our case since calculating the entries of the table requires $O(L^2)$ time and the amount of memory needed for the table is $O(L^2)$.

6.1 Normal Dynamic Programming Algorithm

The implementation of the normal dynamic programming algorithm follows closely the pseudocode of Algorithm 1 in Section 3.1. To improve the performance of the algorithm, the code of the function FINDOPTPATH of Algorithm 1 is directly included in the algorithm, which means the algorithm consists of only one function and no overhead is incurred by function calls. For the trellis structure, a two dimensional array is used, each element consists of a pointer, which is used as back pointer, and a floating point number to store the value of the objective function. The array used for the trellis is shown in Figure 6.1. As shown in Figure 6.1, there are nodes which are never processed and could

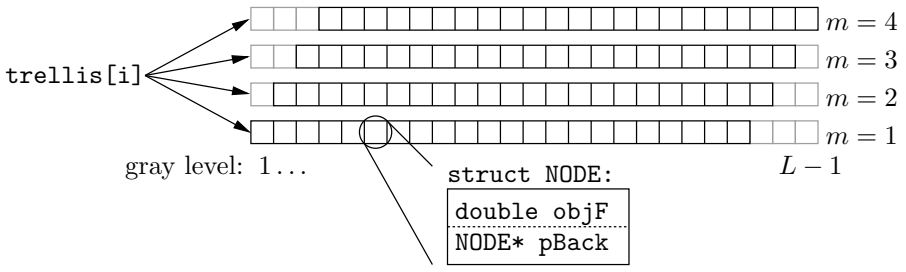

Figure 6.1: Array used to store the trellis structure.

therefore be omitted to save memory, but in order to keep the conversion from the gray level value to array indices simple, the array contains more elements than needed by the algorithm. Like this, the index of the second dimension of the array corresponds directly to the gray level value. The memory needed for the trellis is $(M-1) \times (L-1)$ elements instead of $(M-1) \times (L-M+1)$ elements. The number of additional elements is therefore $M^2 - 3M + 2$. Since M is small compared to L, the memory overhead is not significant.

6.2 DP Combined with Divide-and-Conquer Matrix Searching

The algorithm, which combines dynamic programming and divide-and-conquer matrix searching, uses the same trellis structure as the normal dynamic programming algorithm. The functionality of the matrix search function is essentially the same as the one of Algorithm 3. Passing the submatrix to the recursive call is accomplished by using the indices of the upper left and lower right corners of the submatrix as function arguments:

```
void matrixSearch(int lCornerY, int lCornerX, int rCornerY, int rCornerX);
```

A global parameter is needed to indicate the stage of the trellis, where the matrix search is conducted. It is used inside the function to calculate the trellis indices from the matrix coordinates. In order to further decrease the execution time of the algorithm, the fact that the matrix is lower triangular is exploited. This is done by modifying the search for the maximum in the middle row of the matrix (line 2 in Algorithm 3) to consider only columns $c \leq r$, where c denotes the column and r the row index, respectively.

6.3 DP Combined with SMAWK Matrix Searching

Like the implementation using the divide-and-conquer algorithm, this implementation employs the same array to store the trellis as the normal dynamic programming algorithm. As shown in Section 4.3.2, the SMAWK algorithm can delete columns from the matrix and uses local matrix coordinates throughout the recursions to access the matrix elements. This properties of the algorithm make it difficult to write an efficient implementation using a low-level language such as ANSI C. In fact, only implementations using high-level languages like Java or Python are found on the internet. An other point to consider is, that no dynamic memory allocation is allowed during the runtime of the algorithm, because allocating memory is usually slow and the required time unpredictable. For the ANSI C implementation of the SMAWK algorithm, small changes to the original algorithm introduced in [6] prove to be very helpful. In [6], the authors advocate the use of a linked list, called predecessor array, to delete columns from the matrix. The function REDUCE is modified to work with this linked list and it is shown, that using the modified function leads to an algorithm which has the same time complexity as the original SMAWK algorithm. In our implementation, the REDUCE function is very similar to the function NEW-REDUCE of [6]. Each element of the linked list consists of a integer variable and a pointer. The integer variable is used to store the global column number and the pointer indicates the previous column. The structure of the linked list before and after REDUCE has been executed, is shown in Figure 6.2. The rightmost element of the linked list is a dummy element, it is used by the REDUCE function. The leftmost column is indicated by a pointer which is pointing to `null`. Note, that the elements are stored in an array and therefore are arranged next to each other in memory. This is needed because the list is sometimes accessed like an array. As said before, no dynamic memory allocation is allowed during the execution of the algorithm. Since a linked list is needed at each level of the recursion, memory for multiple lists must be allocated. In order to avoid multiple memory allocations, the memory for all lists together is allocated before the algorithm starts. The total number of elements needed for searching an $m \times n$ matrix is given by (6.6).

$$N_{\text{elements}} = n + 1 + \sum_{i=0}^{\lfloor \log_2(m) \rfloor - 1} \left\lfloor \frac{m}{2^i} \right\rfloor + 1. \tag{6.6}$$

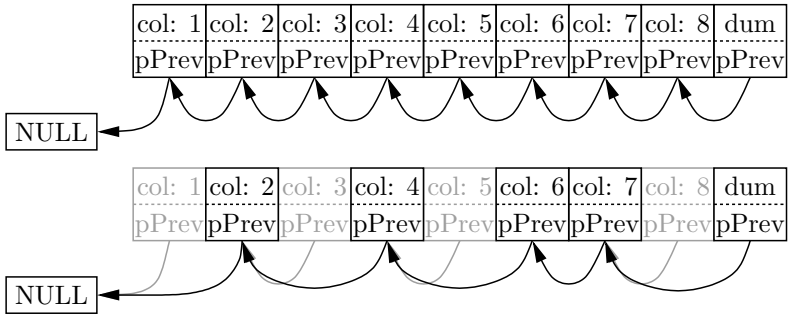

Figure 6.2: Linked list before and after REDUCE.

Therefore, the elements of all lists together are located in one array, which is N_{elements} long. The first $n+1$ elements are used for the initial call, the next $m+1$ for the first recursive call, the next $\lfloor m/2 \rfloor + 1$ for the second recursive call and so on. The structure of this list is illustrated in Figure 6.3. This linked list is a central part of the implementation

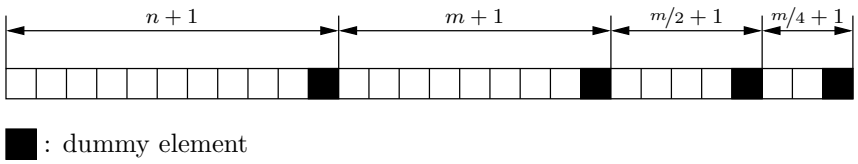

Figure 6.3: All linked list for a 8×10 matrix.

and is used by all functions of the SMAWK algorithm. The prototypes for the SMAWK, REDUCE and MFILL function used in the implementation are the following:

```
void smawk(int m, int n, int rowM, int row0, struct EL* myMatr, struct EL* lstMatr);
struct EL* reduce(int m, int n, int rowM, int row0, EL* myMatr);
void mfill(int m, int rowM, int row0, struct EL* redMatr);
```

The parameters `m` and `n` indicate the number of rows and the number columns of the matrix, respectively. The elements of the linked lists are of type `struct EL`. The parameter `myMatr` points to the leftmost element of the linked list for the current call of the `smawk` function. In the initial call of the `smawk` function, the leftmost $n+1$ elements of the linked list are initialized, the column numbers are set to $1 \ldots n$ and the pointers point to the next element to the left, as shown in Figure 6.2. For the recursive calls, the parameter `lstMatr` is used, it points to the rightmost element of the linked list one recursion level above. The linked list of the recursion level above is traversed by following the pointers and the column numbers are copied into the linked list of the current call. At the same time, the pointers of the linked list are initialized to point to the next element to the left (or to `null` if it is the leftmost element). Since the elements are located next to each other in memory, the linked list can be accessed like an array and initializing the list from right to left without following the pointers, is possible. Indicating whether the call of `smawk` is recursive or not, is accomplished by setting `lstMatr` to `null` in the initial call. After the linked list has been initialized, `reduce` is executed, it also has a parameter `myMatr`, which points to the leftmost element of the current linked list. It deletes $n - m$ elements from the linked list and returns a pointer to the rightmost element, this pointer is later used for the recursive call of `smawk` and for `mfill`. After the last recursive call of `smawk` returns, `mfill` is executed. Since the pointers of the linked list point to the element to the left of

the current element, searching the the matrix form the top left to the lower right corner, like in the original algorithm, would mean all the pointers in the linked list had to be reversed. In order to decrease the execution time, the function `mfill` is modified to search the matrix from the lower right to the top left corner. Therefore, reversing the pointers is not necessary. Every time the algorithm finds a maximum, it stores the column index in an array, which is m elements long (where m is the number of rows of the initial matrix), sets the correct backpointer in the trellis and updates the value of the objective function of the corresponding node. Knowing which node is affected in the trellis is accomplished by knowing the current stage, which is a global variable, and the global row and column indices. The linked list is used for the column indices, for the row indices the parameters `rowM` and `row0` are used. Their names stand for row multiplier and row offset, respectively. The row indices in the implementation go from 0 to $m-1$ instead of 1 to m, but the rows with index $1, 3, 5..$ are considered as even-numbered. For the initial call, `rowM` is one and `row0` is zero. For the recursive call of `smawk`, the current row multiplier is multiplied by two and the the row offest is set to `rowM` + `row0`. The recursive call of `smawk` is given by the following code:

```
smawk(m/2,m,2*rowM,rowM+row0,myMatr + n + 1, redMatr);
```

Where `redMatr` is the pointer returned by `reduce`. By using the row offset and row multiplier, passing only the even-numbered rows is straightforward because the global row number r_{glob} can be directly calculated from the local row number r_{loc}, as shown in (6.7).

$$r_{\text{glob}} = r_{\text{loc}} \cdot \texttt{rowM} + \texttt{row0} \tag{6.7}$$

The global row number is calculated every time the algorithm needs to access an element of the matrix.

7 Execution Time Measurements

In this thesis, three different algorithms for efficient multilevel thresholding have been introduced. Their time complexities of $O(ML^2)$, $O(ML \log L)$ and $O(ML)$ give an upper bound for the execution time of the algorithms. It is clear, that the algorithm which combines dynamic programming and the SMAWK matrix searching algorithm and has a time complexity of $O(ML)$ outperforms the other algorithms if L is sufficiently high. However, from the time complexity alone it is not possible to say which algorithm is the fastest for a certain combination of M and L because the constant factors are unknown. In practice, overhead is incurred by operations such as managing the linked list of the SMAWK algorithm or recursive function calls. Therefore, a theoretical derivation of the actual execution time is very involved and is is difficult to verify the correctness of the result. Instead of trying to calculate the execution times, the implementations for the Otsu criterion are used for performance measurements. Throughout the rest of this chapter, the measurement setup is explained and the results of the measurements are discussed.

7.1 Measurement Setup

When comparing the execution times of the different algorithms, accurate time measurements are crucial. In order to reach a high accuracy, the algorithms are not executed from Matlab (as a mex file) but are included in a standalone application. The application can be run from the command prompt and program options are used to specify which algorithm is used, the file to load the histogram from, and the number of classes. After the algorithm has found the thresholds, the application returns the time which was needed to find the thresholds. Highly accurate time measurements are obtained by running the application with real time priority and disabling paging of the memory pages of the application. The application is run with real time priority by setting the scheduler to round robin scheduling and giving it the highest possible priority (90). Like this, the application is never preempted by another process and the time measured is the actual time needed to find the thresholds. Disabling paging of the memory pages is achieved by locking the pages with the `mlockall` command. The time measurements is started, after all the memory needed by the algorithm has been allocated. At this point, the histogram has already been loaded and the next task is the preprocessing step described in the last chapter. As soon as the algorithm has found all the theresholds, the time measurement is stopped. A Dell Dimension 9100 PC with an Intel Pentium 4 2.8GHz, dual core processor and 2GByte RAM is used for the measurements. The operating system is Linux (Knoppix 4.02, Kernel 2.6.12).

The histogram of the Lenna[1] image (converted to gray scale), and the Fishing Boat[2] are used for the measurements. Since both images only contain 256 gray levels, the histograms are successively interpolated to 512, 1024, 2048,..,2^{20} gray levels. The following equation

[1] http://sipi.usc.edu/database/misc/4.2.04.tiff
[2] http://sipi.usc.edu/database/misc/boat.512.tiff

is used for one interpolation step:

$$h_{\text{new}}(g) = \begin{cases} h_{\text{old}}\left(\dfrac{g+1}{2}\right), & \text{if } g \text{ is odd,} \\ \dfrac{1}{2}h_{\text{old}}\left(\dfrac{g}{2}\right) + \dfrac{1}{2}h_{\text{old}}\left(\dfrac{g}{2}+1\right), & \text{if } g \text{ is even and } g < L_{\text{new}}, \\ h_{\text{old}}(L_{\text{old}}), & \text{if } g = L_{\text{new}}. \end{cases} \tag{7.1}$$

Matlab is used to interpolate the histograms. The interpolated histogarms are normalized ($\sum h(g) = 1$) and stored as binary files. The data type used is double (64bit floating point), this data type is also used for all floating point operations in the implementations of the thresholding algorithms. As a third type of histograms, randomly generated histograms are used. The random histograms are generated using the rand command of Matlab. They have the same sizes as the other histograms and are also normalized and stored as binary files.

All the execution time measurements are executed by a shell script, which runs the thresholding application with the necessary options (algorithm, histogram file, number of classes) and stores the results as text files. The results are evaluated by reading the text files into Matlab.

7.2 Discussion of the Measured Execution Times

From the measured execution times, statements about which is the fasttest algorithm for given combinations of M and L can be made. In the first part of this section, it is shown which algorithm is the most efficient when the thresholds are calculated for images with a small number of gray levels. The execution times for higher numbers of gray levels are discussed in the next part. For some algorithms, the amount of work and therefore their execution time also depends histogram, this effect is shown at the end of this section.

7.2.1 Execution Times for Small Numbers of Gray Levels

The number of gray levels of a normal gray scale image is 256. Therefore, it can be expected that the multilevel thresholding algorithms are mainly used for such images. The execution times of the three different algorithms is shown in Figure 7.1. Obviously,

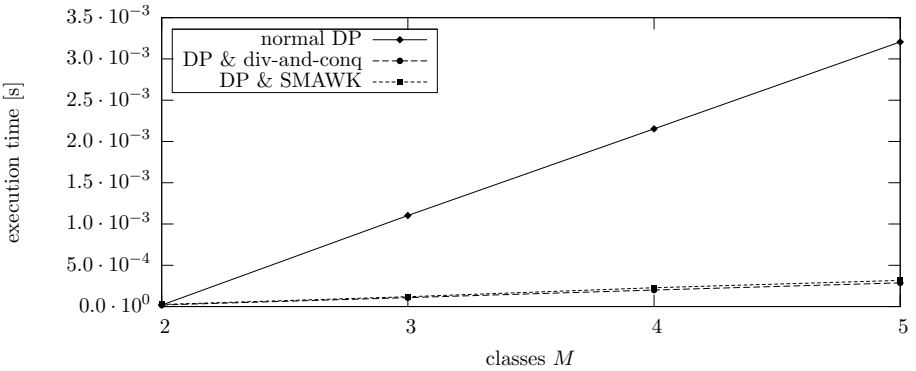

Figure 7.1: Runtimes of the algorithms for $L = 256$ (histogram: Lenna).

all algorithms are several orders of magnitude faster than an exhaustive search [15], when

used for more than two classes. For two classes, the algorithms calculate the value of the objective function for every possible position of the threshold and therefore perform an exhaustive search. It can also be seen from Figure 7.1, that the execution time of all algorithms is proportional to the number of classes. The two algorithms which combine dynamic programming and matrix searching are both about ten times faster than the normal dynamic programming algorithm. The lowest execution times are achieved by the algorithm combining dynamic programming and divide-and-conquer matrix searching, even though it has a higher time complexity than the algorithm which employs SMAWK. An explanation for this may be the overhead incurred by the complex implementation of the SMAWK algorithm. Therefore, for images with only 256 gray levels, the algorithm which uses divide-and-conquer matrix searching is the best choice.

When the number of gray levels is increased, the low time complexity of the SMAWK algorithm causes the algorithm, which employs this matrix searching technique, to become the fastest. This effect can be observed in Figure 7.2. The number of gray levels where

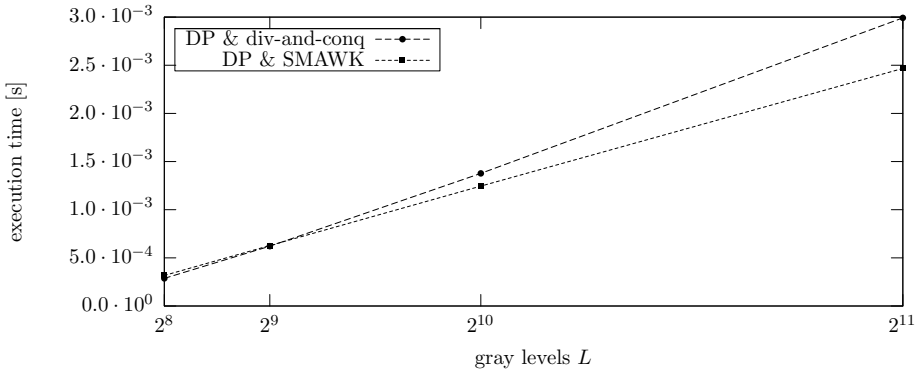

Figure 7.2: Execution times for $L = 256 \ldots 2048$, $M = 5$ (histogram: Lenna).

using SMAWK instead of divide-and-conquer matrix searching becomes advantageous lies somewhere between 512 and 1024. Since the precise number highly depends on the efficiency of the implementations, on the number of classes, and on the histogram it is not shown here. In general, it can be said that the thresholding algorithm which employs SMAWK matrix searching should be used for images with more than 1024 gray levels. Using this algorithm for images with fewer gray levels results in slightly higher execution times than when the combination of dynamic programming and divide-and-conquer matrix searching is employed. Since the execution time difference is very small, using the algorithm which employs SMAWK matrix searching is also a viable choice for images with fewer gray levels.

7.2.2 Execution Times for Higher Numbers of Gray Levels

For some applications, gray scale images with more than 256 gray levels are common. Computer tomography, where images with 14bits per pixel are typical, is an example for such an application. Employing the normal dynamic programming algorithm to find the optimal thresholds in an image with such a high number of gray levels results in very high execution times, as shown in Figure 7.3. The L^2 factor in the time complexity of the normal dynamic programming algorithm becomes clearly visible. When this algorithms is used to segment an image with 2^{16} gray levels into five classes, the execution time is about

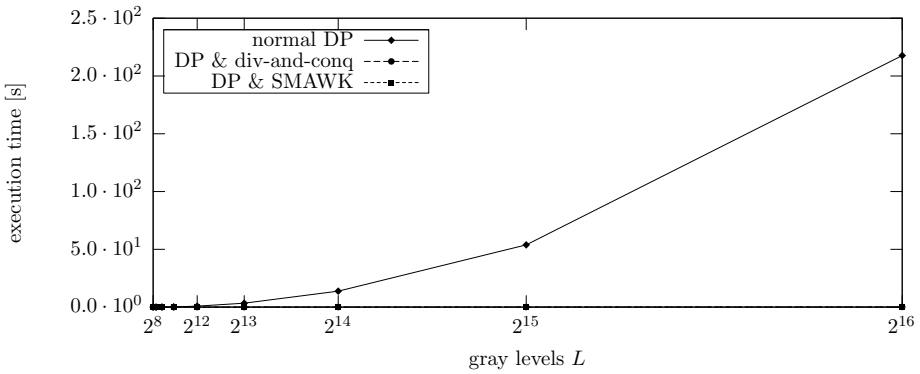

Figure 7.3: Execution times for $L = 2^8 \dots 2^{16}$, $M = 5$ (histogram: Lenna).

217s, while the execution times of the other algorithms are still below 1s. Because of the quadratic factor, the normal dynamic programming algorithm is very slow for images with high numbers of gray levels and one of the other algorithms should be used.

The execution times of the algorithms which combine dynamic programming and matrix searching is shown in Figure 7.4, where the histograms are segmented into 5 classes. It

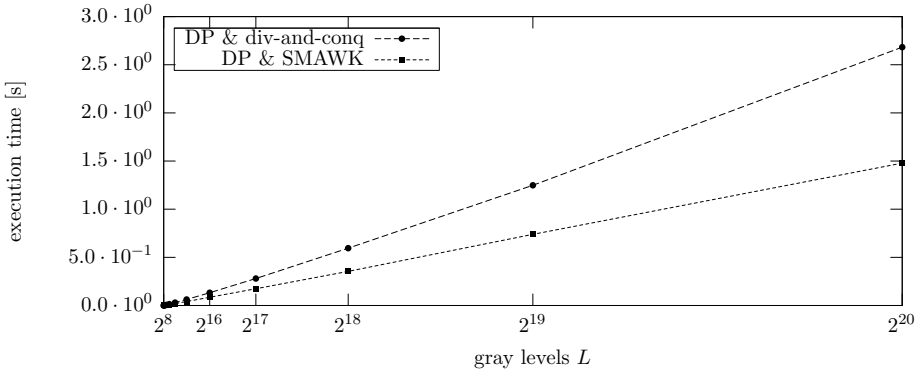

Figure 7.4: Execution times for $L = 2^8 \dots 2^{20}$, $M = 5$ (histogram: Lenna).

can be seen, that the execution time of the algorithm combining dynamic programming and divide-and-conquer matrix searching grows fatser than linear, this is caused by the the $L \log L$ factor in the time complexity of the algorithm. The execution time of the algorithm which employs SMAWK increases linear with the number of gray levels as expected from the $O(ML)$ time complexity. Note, that this algorithm only requires about 1.5s to find the optimal thresholds for 2^{20} gray levels and 5 classes. In contrast, the normal dynamic programming algorithm would require about one hour to find the thresholds (extrapolated from $L = 2^{16}$ and $M = 5$). Using an exhaustive search to find the thresholds becomes literally impossible for such a high number of gray levels and 5 classes, because the objective function had to be calculated $\binom{2^{20}-1}{4} = 5.0 \cdot 10^{22}$ times. Even very fast computer, which is able to calculate and compare the objective function in 1ns, would still require more than one million years to find the thresholds.

7.2.3 Relation between the Histogram and the Execution Time

So far, only interpolated versions of the histogram of the Lenna image have been used for the execution time measurements. By using histograms with the same shape, the matrices which are searched by the matrix searching algorithms always have a similar structure. Therefore, it can be expected that the execution time only depends on the size of the matrix, which means on the number of classes and the number of gray levels. The structure of the matrix only has an influence on the execution times of the algorithms which employ divide-and-conquer or SMAWK matrix searching. The amount of work performed by the normal dynamic programming algorithm or an exhaustive search does not depend on the histogram and therefore only on L and M. The histograms used for the runtime measurements are shown in Figure 7.5. The random histograms contain random

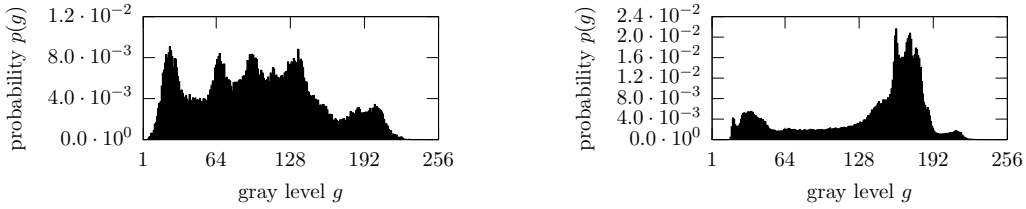

Figure 7.5: Normalized histograms of the Lenna and the Fishing Boat image.

numbers and therefore less structure than the other histograms. From Figure 7.6, it can be seen that the execution time of the normal dynamic programming algorithm does not depend on the structure of the histogram, as expected. The amount of work performed

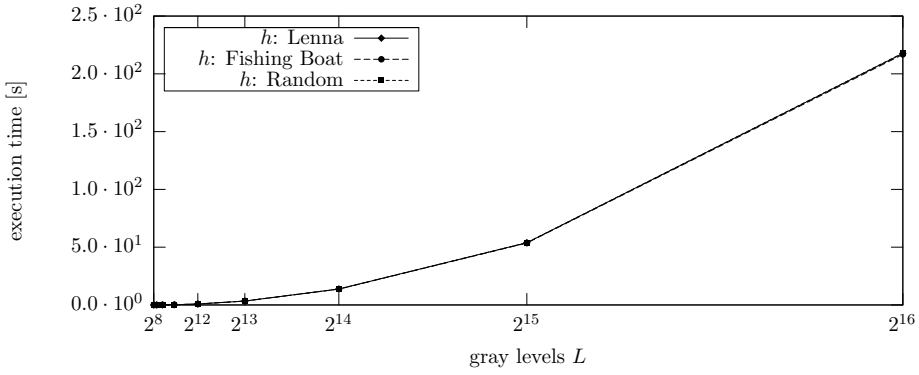

Figure 7.6: Execution times, normal DP algorithm ($M = 5$).

by the efficient matrix searching algorithms depends on the structure of the matrix. In Figure 7.7, the execution times of the algorithm which combines dynamic programming and divide-and-conquer matrix searching are shown. Note, that the algorithm has the highest execution times when random histograms are used. An explanation for this is that the matrices have less structure and this causes more work for the algorithm. Even though the matrix is still totally monotone. The influence of the histogram structure becomes more significant when the algorithm, which combines dynamic programming and SMAWK matrix searching, is used, as shown in Figure 7.8. This can be explained by the fact, that the structure of the matrix is more exploited by the SMAWK algorithm than by

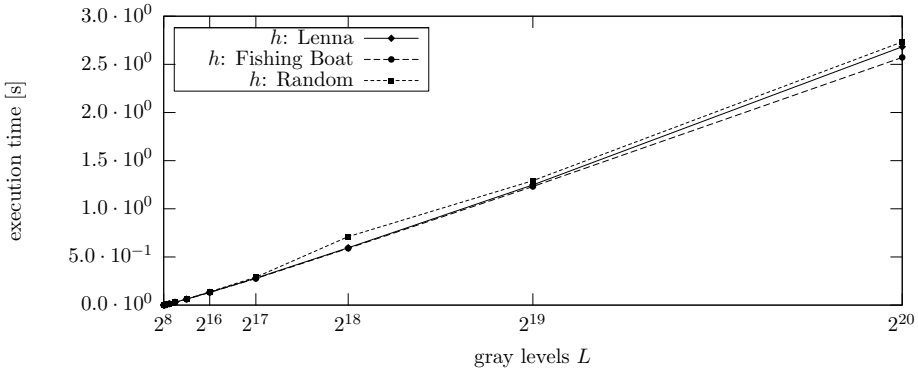

Figure 7.7: Execution times, DP combined with divide-and-conquer algorithm ($M = 5$).

the divide-and-conquer algorithm. In conclusion, it can be said that the structure of the

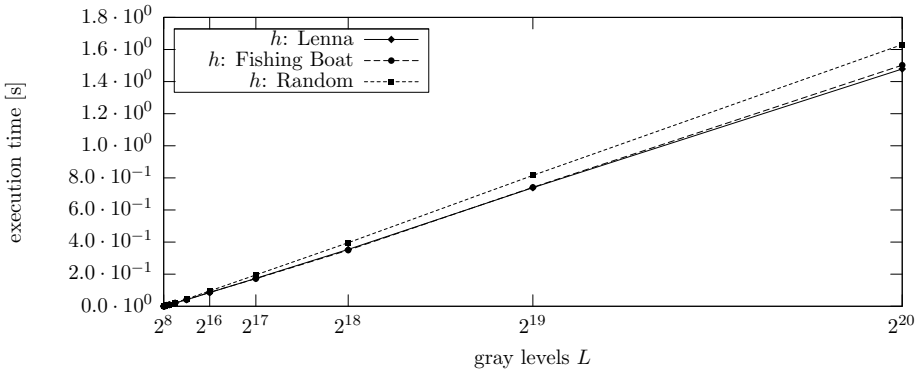

Figure 7.8: Execution times, DP combined with SMAWK algorithm ($M = 5$).

histogram has only a small influence on the execution times of the algorithms. The time required when a random histogram is used, is likely to be close to the worst case execution time. Therefore, this time should be considered when the algorithms are used in a real time system.

8 Automatic Determination of the Best Number of Classes

This chapter treats the independent topic about how to find out, how many classes are present in an image. This is a nontrivial problem, and to find a reasonable solution could be investigated in a separate project. One question every method, which tries to automatically determine the number of classes, has to face is: What is considered as a "good" number of classes? For histograms, which consist of well separated populations (bimodal, multimodal), the right number of classes is obviously the number of distinct modes, but what is a the right number of classes for a histogram with no distinct modes? There is no general answer to this question and depending on the application, the "good" number of classes may differ.

The hypothesis is, that by employing a trellis structure as shown in Figure 8.1 and observing the costs in the last column, one could find a reasonable number of classes. The collection of the costs in the last column is named end cost function EF(m). The

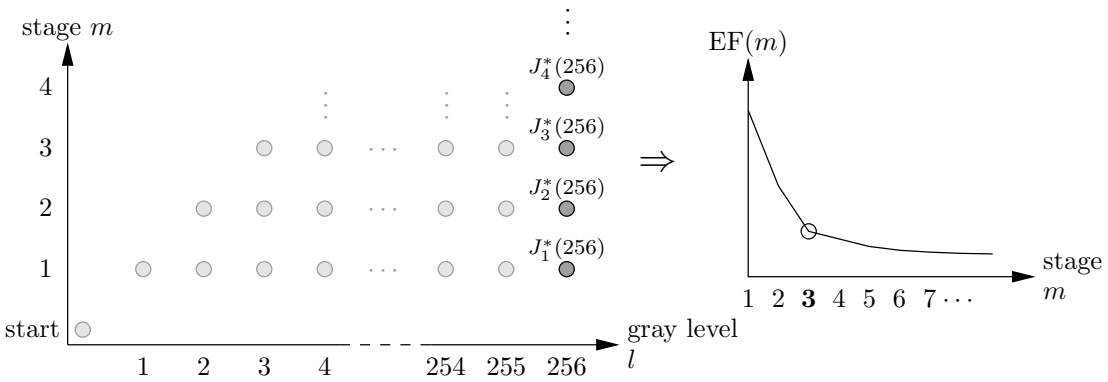

Figure 8.1: Extended trellis structure.

assumption is, that the end cost function shows a significant change at the good number of classes. This idea is investigated for four thresholding criteria Otsu, Kapur, Li and Kittler. We can show, that if the objective function introduced by Kittler and Illingworth is used, the end cost function EF(m) gives useful information about the number of classes present in an image.

In Section 8.1, the end cost functions for the different criteria are compared. Furthermore, it is shown, that the end cost function for the Kittler criterion contains more information than the end cost function from the other criteria, and reasons are discussed. For the Kittler criterion, two methods to automatically find the best number of classes are presented and their performance is compared.

8.1 Observed Methods

The end cost functions for the Otsu, Kapur, Li and the Kittler criterion have been evaluated for several histograms, including real image histograms as well as generated histograms, composed of Gaussian distributions. Two of the images used are shown in Figure 8.2, and their histograms are presented in Figure 8.3A)[1] and 8.3B)[1].

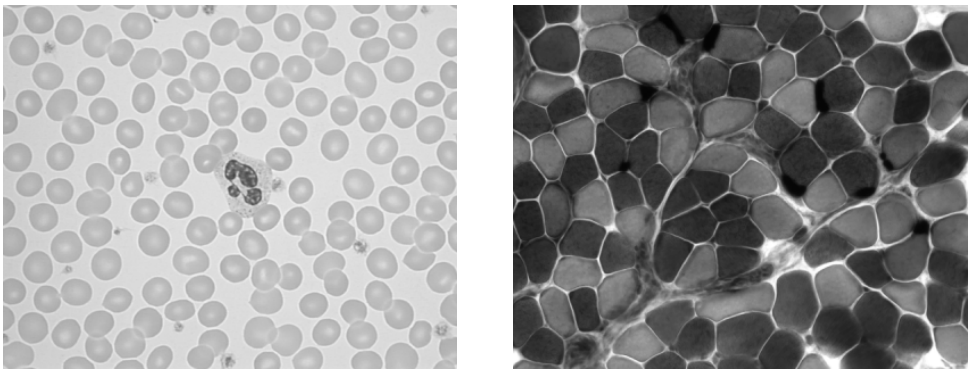

Figure 8.2: Test images: a) blood cell image, b) muscle fibres.

In Figure 8.3, the different end cost functions are plotted. It can be observed that for Kittler, Otsu and Li the end cost funcitons usually show some distinct corners. Together with the results for other histograms, it can be seen that the Kittler end cost function (KEF) is a better indicator, than the end cost functions for Otsu and Li. For Kapur, no useful relation between end cost function and a good number of classes can be observed. A know drawback of the Otsu method is, that it works poorly for the case, when the number of pixels in the populations are extremely different. On the other hand, the method by Kittler selects reasonable thresholds even when variances or sizes of the populations are different [18]. Furthermore, the Kittler method is considered as the best performing thresholding algorithm in survey [1], where it outperforms all other methods including the Otsu, Kapur and Li method.

8.2 Two Methods to find the Number of Classes

In order to find the best number of classes from the Kittler end cost function (KEF), two methods are investigated. Following, both methods are explained an their performance is shown.

8.2.1 Method 1: Second Derivative

This method exploits the fact, that the Kittler end cost function usually indicates a good number of classes with a distinct corner. The corner can be located by taking the second derivative of the end cost function. Therefore, the number of classes can be found as

$$NoC = \arg\max_{m} \left(\frac{d^2 \mathrm{KEF}(m)}{dx^2} \right) + 1, \tag{8.1}$$

[1] Source: http://cclcm.ccf.org/vm/

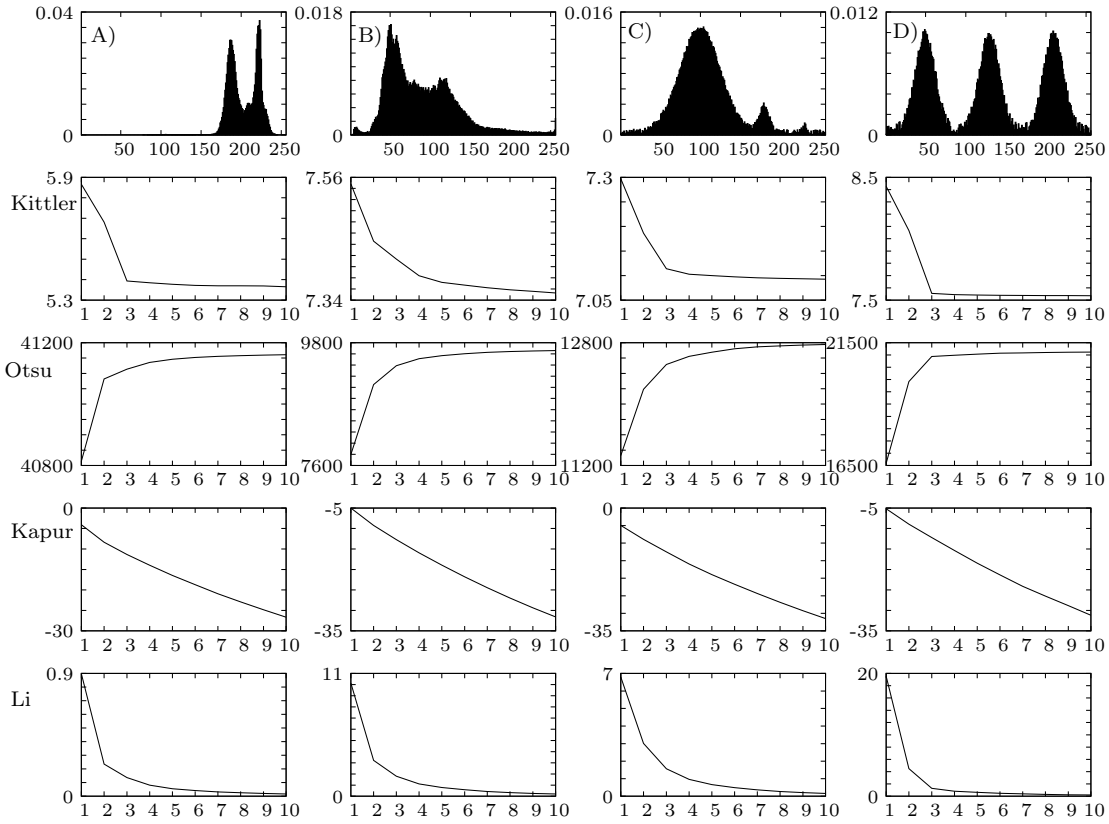

Figure 8.3: Comparison of the end cost functions for different histograms. A) histogram of 8.2a), B) histogram of 8.2b).

where $\arg\max_m\{\}$ returns the position of the maximal value. The performance of this method is shown in Figure 8.4, where the gray dashed lines show the thresholds set by the Kittler algorithm for the determined number of classes.

8.2.2 Method 2: Difference of $\mathrm{KEF}(m)$ and $\mathrm{FEF}_\alpha(m)$

This method searches for the farthest distance between the Kittler end cost function for a flat histogram, called FEF, and the KEF for the observed histogram as shown in Figure 8.5. Since the placement of the thresholds for flat histogram is predictable, an equation for the Kittler end cost function for a flat histgram can be derived. Like this, the KEF for a flat histogram can be calculated directly. Otherwise, the end costs had to be calculated by running the Kittler thresholding algorithm and extracting them from the trellis structure (see Figure 8.1). The $\mathrm{FEF}_\alpha(m)$ formula is quite complicated, but can be derived as

$$\begin{aligned} \mathrm{FEF}_\alpha(m) &= 4\frac{\alpha(m-1)}{L}\log\left(\frac{L}{4}\right) + \frac{k(m)}{L}\log\left(\frac{\frac{1}{12}(k^2(m)-1)}{\left(\frac{k(m)}{L}\right)^2}\right), \\ k(m) &= L - 2\alpha(m-1), \end{aligned} \tag{8.2}$$

where α is a parameter which changes the slope of the function as shown in Figure 8.5.

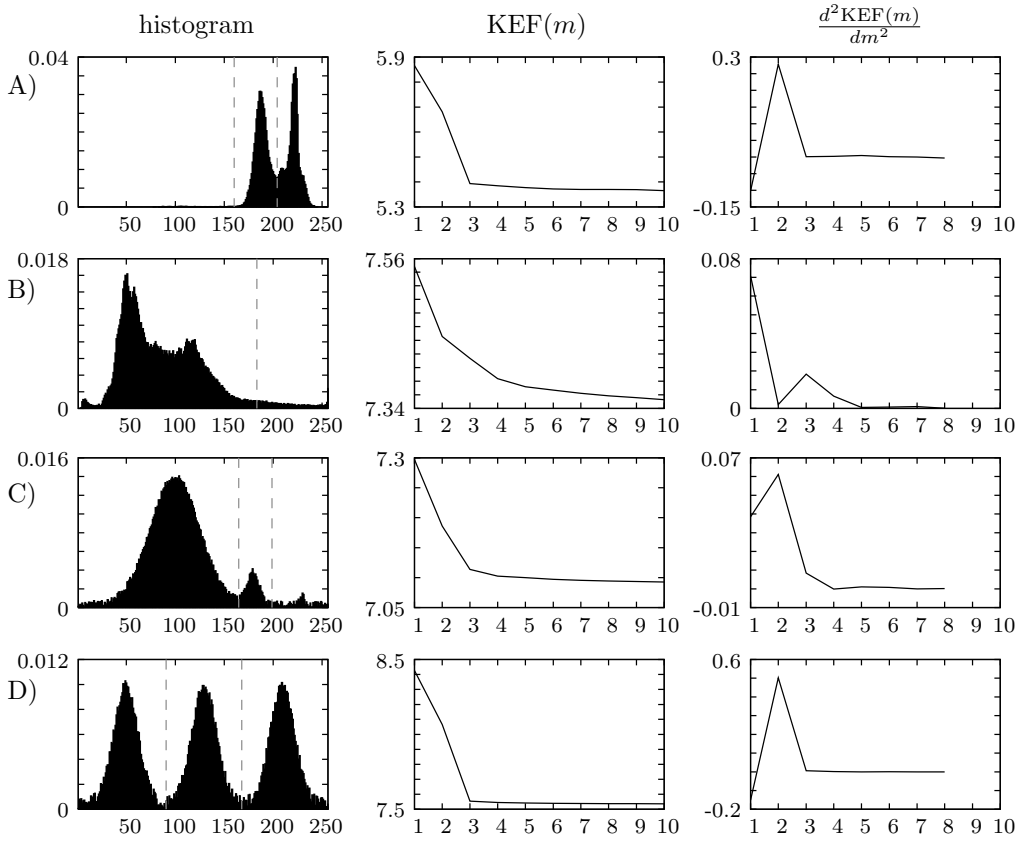

Figure 8.4: Performance of the Method 1.

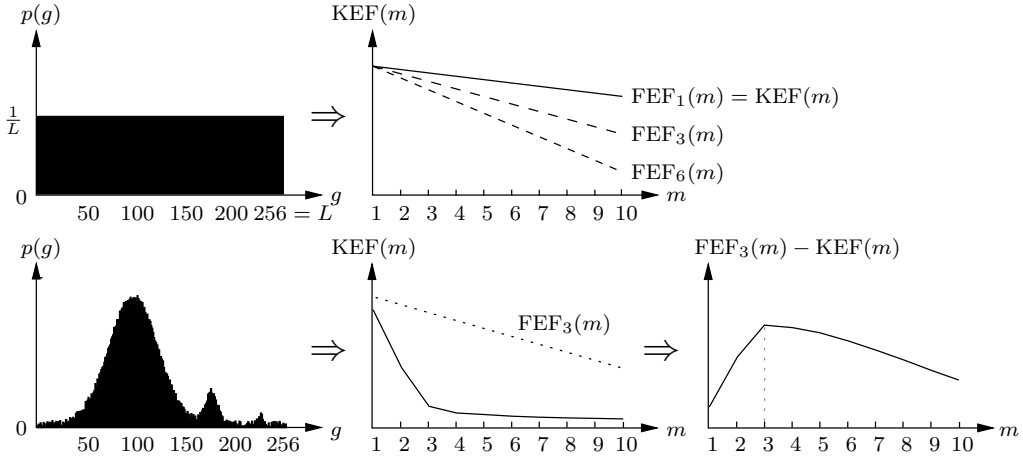

Figure 8.5: Principle mode of operation of Method 2.

In Figure 8.6 the performance of Method 2 is shown, where the results of two different slopes $\alpha = 1$ and $\alpha = 7$ plotted. For $\alpha = 7$ this method gives quite satisfying results throughout the whole test set.

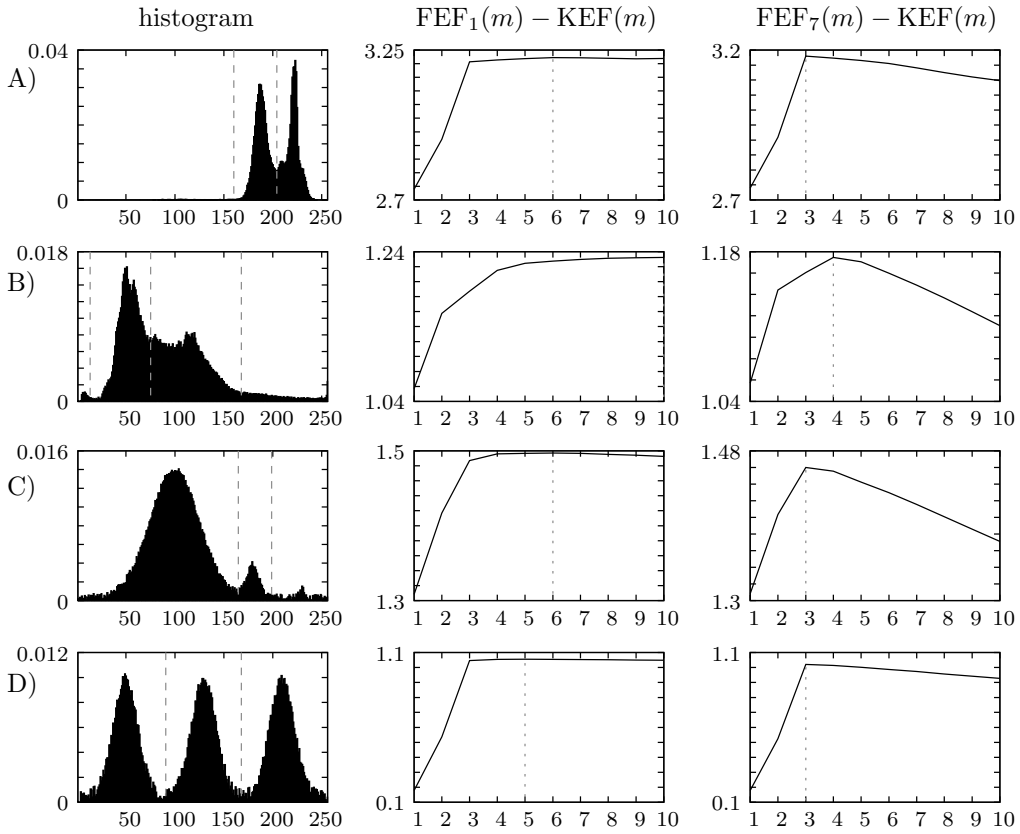

Figure 8.6: Performance of Method 2.

8.3 Discusion

The determination of a good number of classes is not a trivial task, since even for a human it is hard to tell how many classes are present in an image when there are no distinct populations visable in the histogram. Therefore, if the number of classes is found by analyzing the histogram, it is reasonable to constrain the test set to images, which have distinct modes in their histograms.

By taking all test results into account, one can observe that the Kittler end cost function gives the most reliable information about the number of classes. If Method 2 is used as described in Section 8.2.2 with α from around $5 \ldots 10$, the determined numbers of classes are quite satisfying. Compared to Method 1 (see Section 8.2.1), Method 2 does not need a distinct corner and provides also reasonable results for histograms which are not well separated. The Method 2 has an other advantage, if an image is thresholded with an algorithm using Method 2 to automatically find the number of classes, the algorithm can be stopped as soon as the difference $\mathrm{FEF}_\alpha(m) - \mathrm{KEF}(m)$ decreases. In contrary, for an algorithm using Method 1, the algorithm has to be run for a maximal number of classes, followed by the search for the maximum. This is because the second derivative of the end cost function may have local maxima.

For histograms, where the populations are almost of equal size, also the Otsu and the Li end cost function could be employed, altough just for histograms with just a few (< 5) distinct modes. It can be observed, that for artificial histograms with well separated equal

sized Gaussian populations, the corner in the end cost function becomes less distinct as the number of Gaussian modes increases. This confirms the statement in the original paper of Otsu [10], that the selected thresholds generally become less meaningful as the number of classes increases.

For the Kapur criterion, no useful relation between a good number of classes and the end cost function is seen.

9 Conclusion

In our thesis, we identify a class of objective functions for which the optimal thresholds can be found by efficient algorithms. We also show, that the optimal thresholds for the well known method proposed by N. Otsu [10] can be found by these algorithms. However, this result is not entirely new, since the method proposed by N. Otsu optimizes an objective function, which is also encountered in the design of an optimal scalar quantizer. The efficient algorithm presented in this thesis have already been proposed by X. Wu for the design of optimal scalar quantizers [5][4]. Also, the connection between the multilevel thresholding, as proposed by N. Otsu, and scalar quantization has been made before [16]. However, the only optimal and efficient multilevel thresholding algorithm for the Otsu method proposed so far, is the dynamic programming algorithm by N. Otsu[13], which has a time complexity of $O(ML^2)$. Even though this algorithm is much more efficient than an exhaustive search, calculating optimal thresholds for images with high numbers of gray levels takes a long time. With the algorithms presented in this thesis, it is possible to find the optimal thresholds much faster.

Furthermore, we find that another method, called minimum cross entropy [11], is a member of the identified class. Therefore, the optimal thresholds for this method can be found in $O(ML)$ time. For an other method [9], we propose the use of the dynamic programming algorithm with a time complexity of $O(ML^2$.

By comparing the execution times of actual implementations, we can make quantitative statements about the efficiency of the algorithms. The measured execution times are consistent with the theoretically derived time complexities. For the measurements, the SMAWK algorithm [3] has been implemented using ANSI C. It is an interesting fact, that the use of the SMAWK algorithm has been proposed for numerous problems, but no signs of the existence of efficient implementations can be found. Therefore, it is probably the first time the SMAWK algorithm has been implemented using a low-level language, such as ANSI C.

A topic, which is treated separately in this thesis, is the question how to determine the number of classes present in an image. We propose the use of a dynamic programming algorithm with a special trellis structure, the total cost for different numbers of classes can be used as an indicator for the best number of classes. The method shows some promising results. However, the method is not yet very sophisticated.

The question of whether or not the thresholds found by the algorithms are meaningful for the segmentation of images is not addressed in our thesis. Finding an objective function which is a member of the identified class and which outperforms current thresholding methods could be the topic of further research.

Bibliography

[1] Mehmet Sezgin and Bülent Sankur, "Survey over image thresholding techniques and quantitative performance evaluation," *Journal of Electronic Imaging*, vol. 13, no. 1, pp. 146–165, Jan. 2004.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, MIT Press, Cambridge, MA, USA, 2001.

[3] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber, "Geometric applications of a matrix-searching algorithm.," *Algorithmica*, vol. 2, pp. 195–208, 1987.

[4] Xiaolin Wu and Jon G. Rokne, "An o(kn lgn) algorithm for optimum k-level quantization on histograms of n points.," in *ACM Conference on Computer Science*, 1989, pp. 339–343.

[5] Xiaolin Wu and Kaizhong Zhang, "Quantizer monotonicities and globally optimal scalar quantizer design.," *IEEE Transactions on Information Theory*, vol. 39, no. 3, pp. 1049–, 1993.

[6] John Hershberger and Subhash Suri, "Matrix searching with the shortest-path metric.," *SIAM J. Comput.*, vol. 26, no. 6, pp. 1612–1634, 1997.

[7] Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf, "Perspectives of monge properties in optimization.," *Discrete Applied Mathematics*, vol. 70, no. 2, pp. 95–161, 1996.

[8] Amir Said, "On the reduction of entropy coding complexity via symbol grouping: Part i - redundancy analysis and optimal alphabet partition.," Technical report, Aug. 2004.

[9] J.N. Kapur, P.K. Sahoo, and A.K.C. Wong, "A new method for gray-level picture thresholding using the entropy of the histogram," *Computer Vision, Graphics, and Image Processing*, vol. 29, pp. 273–285, 1985.

[10] Nobuyuki Otsu, "A threshold selection method from gray level histograms," *IEEE Trans. Systems, Man and Cybernetics*, vol. 9, pp. 62–66, Mar. 1979.

[11] Chun Hung Li and C. K. Lee, "Minimum cross entropy thresholding," *Pattern Recognition*, vol. 26, no. 4, pp. 617–625, Apr. 1993.

[12] Xiping Luo and Jie Tian, "Multi-level thresholding: Maximum entropy approach using icm," in *ICPR '00: Proceedings of the International Conference on Pattern Recognition (ICPR'00)-Volume 3*, Washington, DC, USA, 2000, p. 3786, IEEE Computer Society.

[13] Nobuyuki Otsu, "An automatic threshold selection method based on discriminant and least squares criteria," *ransactions of the IECE of Japan*, vol. 64-D(4), pp. 349–356, 1980.

[14] Joel Max, "Quantizing for minimum distortion," *IRE Transactions on Information Theory*, vol. IT-6, pp. 7–12, Mar. 1960.

[15] Ping-Sung Liao, Tse-Sheng Chen, and Pau-Choo Chung, "A fast algorithm for multilevel thresholding," *Journal of Information Science and Engineering*, vol. 17, pp. 713–727, Sept. 2001.

[16] Olli Virmajoki and Pasi Fränti, "Fast pairwise nearest neighbor based algorithm for multilevel thresholding," *Journal of Electronic Imaging*, vol. 12(4), pp. 648–659, Oct. 2003.

[17] J. Kittler and J. Illingworth, "Minimum error thresholding," *Pattern Recogn.*, vol. 19, no. 1, pp. 41–47, 1986.

[18] Takio Kurita, Nobuyuki Otsu, and Nabih N. Abdelmalek, "Maximum likelihood thresholding based on population mixture models.," *Pattern Recognition*, vol. 25, no. 10, pp. 1231–1240, 1992.