Optimal Decision Trees Generation from *OR*-Decision Tables

Costantino Grana, Manuela Montangero, Daniele Borghesani, and Rita Cucchiara

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Modena e Reggio Emilia
costantino.grana@unimore.it,manuela.montangero@unimore.it,daniele.borghesani@unimore.it,rita.cucchiara@unimore.it

Abstract. In this paper we present a novel dynamic programming algorithm to synthesize an optimal decision tree from *OR*-decision tables, an extension of standard decision tables, which allow to choose between several alternative actions in the same rule. Experiments are reported, showing the computational time improvements over state of the art implementations of connected components labeling, using this modelling technique.

Keywords: Decision tables, Decision trees, Connected components labeling, Image processing

1 Introduction

A decision table is a tabular form that presents a set of conditions which must be tested and a list of corresponding actions to be performed. Each combination of condition entries (*condition outcomes*) is paired to an *action entry*. In the action entries, a column is marked, for example with a "1", to specify whether the corresponding action is to be performed. The interpretation of a decision table is straightforward: all actions marked with 1s in the action entries vector should be performed if the corresponding outcome is obtained when testing the conditions [8]. In general, decision tables are used to describe the behavior of a system whose state can be represented as a vector, i.e. the outcome of testing certain conditions. Given a particular state, the system evolves by performing a set of actions that depend on the given state of the system. The state is described by a particular rule, the action by the corresponding row of the table.

Even if decision tables are easy to read and understand, their straightforward use might not be efficient. In fact, decision tables require all conditions to be tested in order to select the corresponding actions to be executed, and testing the conditions of the decision table has a cost which is related to the number of conditions and to the computational cost of each test. There are a number of cases in which not all conditions must be tested in order to decide which action has to be performed, because the specific values assumed by a subset of conditions might be enough to pick a decision. This observation suggests that

the order with which the conditions are verified impacts on the number of tests required, thus on the total cost of testing. Moreover, after selecting the first condition to be tested, the second condition to test might vary according to the outcome of the first test. What is thus obtained is a *decision tree*, a tree that specifies the order in which to test the conditions, given the outcome of previous tests. Changing the order in which conditions are tested might lead to more or less tests to be performed, and hence to a higher or lower execution cost. The *optimal decision tree* is the one that requires, on average, the minimum cost when deciding which actions to execute.

In a large class of image processing algorithms the output value for each image pixel is obtained from the value of the pixel itself and some of its neighbors. We can refer to these as *local* or *neighborhood* algorithms. In particular, for binary images, we can model local algorithms by means of a limited entry decision table, in which the pixels values are conditions to be tested and the output is chosen by the action corresponding to the conditions outcome.

In this paper we will address the recently introduced *OR*-decision tables [2], providing a dynamic programming algorithm to derive optimal decision trees for such decision tables. The algorithm is applied to the currently fastest connected components labeling algorithm, showing how the proposed optimal synthesis improves, in terms of efficiency, the previous approaches.

2 Decision Tables and Decision Trees

Given a set of conditions and the corresponding actions to be performed according to the outcome of these testing conditions, we can arrange them in a tabular form $\mathcal{T}$ called a *decision table*: each row corresponds to a particular outcome for the conditions and is called *rule*, each column corresponds to a particular action to be performed. A given entry $\mathcal{T}[i, j]$ of the table is set to one if the action corresponding to column j must be performed given the outcome of the testing condition as in row i; the entry is set to zero otherwise. Different rules might have different probability to occur and testing conditions might be more o less expensive to test. The order in which conditions are tested might be influent or not.

There are different kind of decision tables, according to the system they describe: a table is said to be a *limited entry* decision table [6] if the outcome of the testing conditions is binary; it is called *extended entry* otherwise. A table in which there is a row for every possible rule is said to be an *expanded* decision table, and *compressed* otherwise. We will call a decision table an *AND*-decision table if **all** the actions in a row must be executed when the corresponding rule occurs, instead we will call it an *OR*-decision table if **any** of the actions in a row might be executed when the corresponding rule occurs. In particular, *OR*-decision tables were firstly introduced by Grana *et al.* [2].

There are two main approaches to derive decision trees from decision tables: a top-down approach, in which a rule of the table is selected to be the root of the tree and the subtrees are build by recursively solving the problem on the

two portions of the table that are obtained by removing the rule that has been placed in the root; a bottom-up approach, in which rules are grouped together if the associated actions can be decided by testing a common subset of conditions. We adopt the second approach, mainly because it has been shown to be able to handle tables having a larger number of conditions (hence, rules).

Schumacher *et al.* [8] proposed a bottom-up Dynamic Programming technique which guarantees to find the optimal decision tree given an expanded limited entry decision table, in which each row contains only one non-zero value. This strategy can be extended also to limited entry *AND*-decision tables. Lew [5] gives a Dynamic Programming approach for the case of extended entry and/or compressed *AND*-decision tables. In this paper, we extend Schumacher's approach to *OR*-decision tables.

2.1 Preliminaries and Notation

An OR-decision table is described by the following sets and parameters:

- $C = \{c_1, \ldots, c_L\}$ boolean conditions; the cost of testing condition c_i is given by $w_i > 0$;
- $R = \{r_1, \ldots, r_{2^L}\}$ rules; each rule is a boolean vector of L elements, element i corresponding to the outcome of condition c_i; the probability that rule r occurs is given by $p_r \geq 0$;
- $A = \{a_1, \ldots, a_M\}$ actions; rule r_i is associated to a non empty subset $A^{(i)} \subseteq A$ to be executed when the outcome of conditions c_j identifies rule r_i.

We wish to determine an efficient way to test conditions $c_1, \ldots, c_L$ in order to decide *as soon as possible* which action should be executed according to the values of conditions c_j. In particular, we wish to determine in which order conditions c_i have to be checked, so that the minimum number of tests are performed. Such information are given in the form of a tree, called decision tree, and here we will give an algorithm to find the optimal one, intuitively the one that stores these information in the most succinte way.

In the following we will call set $K \subseteq R$ a *k-cube* if it is a subset of rules in which the value of $L - k$ conditions is fixed. It will be represented as a L-vector of k dashes $(-)$ and $L - k$ values 0's and 1's. The set of positions containing dashes will be denoted as D_K. We associate to cube K a set of rules, denoted by A_K, that contains the intersection of the sets of actions associated to the rules in K (might be an empty set). The occurrence probability of the k-cube K is the probability P_K of any rule in K to occur, *i.e.* $P_K = \sum_{r \in K} p_r$. Finally, we will denote with $\mathcal{K}_k$ the set of the k-cubes, for $k = 0, \ldots, L$.

A *Decision Tree* for K is a binary tree T with the following properties:

1. Each leaf ℓ corresponds to a k-cube of rules, denoted by K_ℓ, that is a subset of K. Each leaf ℓ is associated to the set of actions A_{K_ℓ} associated to cube K_ℓ. Each internal node is labeled with a testing condition $c_i \in C$ such that there is a dash at position i in the vector representation of K. Left (resp. right) outgoing edges are labeled with 0 (resp. 1).

2. Two distinct nodes on the same root-leaf path can not be labeled with the same testing condition. Root-leaf paths univocally identify, by means of node and edges labels, the set of rules associated to leaves: conditions labeling nodes on the path must be set to the value of the label on the corresponding outgoing edges, the remaining conditions are set to a dash.

The cost of making a specific decision, is the cost of testing the conditions on a root-leaf path, in order to execute one of the actions associated to the leaf. On average, the cost of making decisions is given by the sum of the root-leaf paths weighted by the probabilty that the rules associated to leaves occur; *i.e.* the average cost of a decision tree is a measure of the cost of testing the conditions that we need to check in order to decide which actions to take when rules occur, weighted by the probability that rules occur.

We formally define the notions of cost and gain of decision trees on cubes, that will be used in the following.

Definition 1 (Cost and Gain of a Decision Tree). *Given a k-cube K, with dashes in positions D_K, and a decision tree T for K,* cost *and* gain *of T are defined in the following way:*

$$cost(T) = \sum_{\ell \in \mathcal{L}} \left(P_\ell \sum_{c_i \in path(\ell)} w_i \right), \tag{1}$$

where $\mathcal{L}$ is the set of leaves of the tree and the w_is are the costs of the testing conditions on the root-leaf path leading to $\ell \in \mathcal{L}$, denoted by $path(\ell)$;

$$gain(T) = P_K \sum_{i \in D_K} w_i - cost(T), \tag{2}$$

where $P_K \sum_{i \in D_K} w_i$ is the maximum possibile cost for a decision tree for K.

An Optimal Decision Tree *for k-cube K is a decision tree for the cube with minimum cost (might not be unique) or, equivalently, with maximum gain.*

Observe that, when the probabilities of the rules in the leaves of the tree sum up to one, the cost defined in equation (1) is exaclty the quantity that we wish to minimize in order to find a decision tree of minimum average cost for the L-cube that describes a given OR-Decision table.

A simple algorithm to derive a decision tree for a k-cube K works recursively in the following way: select a condition index i that is set to a dash and make the root of the tree a node labeled with condition c_i. Partition the cube K into two cubes $K_{i,0}$ and $K_{i,1}$ such that condition c_i is set to zero in $K_{i,0}$ and to one in $K_{i,1}$. Recursively build decision trees for the two cubes of the partition, then make them the left and right children of the root, respectively. Recursion stops when all the rules in the cube have at least one associated action in common.

The cost of the outcoming tree is strongly affected by the order used to select the index that determines the cube partition. In the next section we give a dynamic programming algorithm that determines a selection order that produces a tree with maximum gain.

2.2 Dynamic Programming Algorithm

An optimal decision tree can be computed using a generalization of the Dynamic Programming strategy introduced by [8], with a bottom-up approach: staring from 0-cubes and for increasing dimension of cubes, the algorithm computes the gain of all possible trees for all cubes and keeps track of only the ones having maximum gain. The trick that allows to choose the best action to execute is to keep track of the intersection of actions sets of all the rules in the k-cube. It is possible to formally prove both the correctness and optimality of the algorithm.

1: **for** $K \in R$ **do** ▷ Inizialization of 0-cubes in $R \in \mathcal{K}_0$
2: $Gain^*_K \leftarrow 0$
3: $A_K \leftarrow$ set of actions associated to rule in K
4: **end for**
5: **for** $n \in [1, L]$ **do** ▷ for all possible cube dimentions > 0
6: **for** $K \in \mathcal{K}_n$ **do** ▷ for all possible cubes with n dashes
▷ compute current cube probability and set of actions
7: $P_K \leftarrow P_{K_{j,0}} + P_{K_{j,1}}$ ▷ where j is any index in D_K
8: $A_K \leftarrow A_{K_{j,0}} \cap A_{K_{j,1}}$
▷ compute gains obtained by ignoring one condition at the time
9: **for** $i \in D_K$ **do** ▷ for all positions set to a dash
10: **if** $A_K \neq \emptyset$ **then**
11: $Gain_K(i) \leftarrow w_i P_K + Gain^*_{K_{i,0}} + Gain^*_{K_{i,1}}$
12: **else**
13: $Gain_K(i) \leftarrow Gain^*_{K_{i,0}} + Gain^*_{K_{i,1}}$
14: **end if**
15: **end for**
▷ keep the best gain and its index
16: $i^*_K \leftarrow \arg\max_{i \in D_K} Gain_K(i)$
17: $Gain^*_K \leftarrow Gain_K(i^*_K)$
18: **end for**
19: **end for**
20: BuildTree($K \in \mathcal{K}_L$) ▷ Recursively build tree on entire set of rules

21: **procedure** BuildTree(K)
22: **if** $A_K \neq \emptyset$ **then**
23: CreateLeaf(A_K)
24: **else**
25: $left \leftarrow$ BuildTree($K_{i^*_K,0}$)
26: $right \leftarrow$ BuildTree($K_{i^*_K,1}$)
27: CreateNode($c_{i^*_K}, left, right$)
28: **end if**
29: **end procedure**

Figure 1 reports two examples of the algorithm steps with $L = 3$. **order** is the number of dashes in the cubes under consideration; the portion of a table characterized by the same order referes to one run of the **for** cycle in lines 5 -

order	a	b	c	A	P	G1	G2	G3
0	0	0	0	1	0,125			
	0	0	1	3	0,125			
	0	1	0	1,3	0,125			
	0	1	1	3	0,125			
	1	0	0	2	0,125			
	1	0	1	3	0,125			
	1	1	0	2,4	0,125			
	1	1	1	4	0,125			
1	-	0	0	0	0,250	0,000		
	-	0	1	3	0,250	0,250		
	-	1	0	0	0,250	0,000		
	-	1	1	0	0,250	0,000		
	0	-	0	1	0,250	0,250		
	0	-	1	3	0,250	0,250		
	1	-	0	2	0,250	0,250		
	1	-	1	0	0,250	0,000		
	0	0	-	0	0,250	0,000		
	0	1	-	3	0,250	0,250		
	1	0	-	0	0,250	0,000		
	1	1	-	4	0,250	0,250		
2	-	-	0	0	0,500	0,500	0,000	
	-	-	1	0	0,500	0,250	0,250	
	-	0	-	0	0,500	0,000	0,250	
	-	1	-	0	0,500	0,500	0,000	
	0	-	-	0	0,500	0,250	0,500	
	1	-	-	0	0,500	0,250	0,250	
3	-	-	-	0	1,000	0,750	0,750	0,750

(a)

order	a	b	c	A	P	G1	G2	G3
0	0	0	0	1	0,100			
	0	0	1	3	0,100			
	0	1	0	1,3	0,100			
	0	1	1	3	0,100			
	1	0	0	2	0,100			
	1	0	1	3	0,200			
	1	1	0	2,4	0,100			
	1	1	1	4	0,200			
1	-	0	0	0	0,200	0,000		
	-	0	1	3	0,300	0,300		
	-	1	0	0	0,200	0,000		
	-	1	1	0	0,300	0,000		
	0	-	0	1	0,200	0,200		
	0	-	1	3	0,200	0,200		
	1	-	0	2	0,200	0,200		
	1	-	1	0	0,400	0,000		
	0	0	-	0	0,200	0,000		
	0	1	-	3	0,200	0,200		
	1	0	-	0	0,300	0,000		
	1	1	-	4	0,300	0,300		
2	-	-	0	0	0,500	0,400	0,000	
	-	-	1	0	0,500	0,200	0,300	
	-	0	-	0	0,400	0,000	0,300	
	-	1	-	0	0,600	0,500	0,000	
	0	-	-	0	0,400	0,200	0,400	
	1	-	-	0	0,600	0,300	0,200	
3	-	-	-	0	1,000	0,700	0,800	0,700

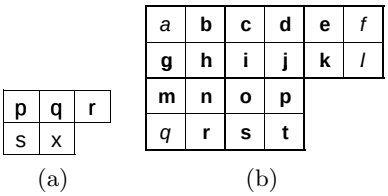
(b)

Fig. 1. Tabular format listing of the algorithm steps.

19. **a,b** and **c** are the testing conditions, A (resp. P) the set of actions (resp. probability occurrence) associated to the cube identified by the values assumed by conditions on a particular row of the table. **Gi**, with $i = 1, 2, 3$, are the gains computed by the successive runs of the for cycle in lines 9 - 15. The highlighted values are those leading to the optimal solution (line 17). In case (a), a set of actions and alternatives with uniform probabilities is shown. The action sets get progressively reduced up to empty set, while the probabilities increase up to unity. While the n-cube order increases, different dashes can be selected (thus different conditions may be chosen) and the corresponding gain is computed. In this particular case, any choice of the first condition to be checked is equivalent (we conventionally choose the leftmost one). When the probability distribution of the rules changes, as in case (b), the gains change accordingly and, in the depicted case, a single tree (rooted with condition b) is the optimal one.

2.3 Computational time.

The total number of cubes that are analyzed by the algorithm is 3^L, one for all possible words of length L on the three letter alphabet $\{0, 1, -\}$. For each cube K of dimension n it computes: (1) the intersection of the actions associated to the cubes in one partition (line 8); this task that can be accomplished, in the



(a) (b)

Fig. 2. The standard mask used for raster scan connected components labeling since original's Rosenfeld approach (a), and the 2×2 block based mask used in our novel approach. In both masks, non-bold letters identify pixels which are not employed in the algorithm.

worst case, in time linear with the number of actions. (2) The gain of $n \leq L$ trees, one for each dash (lines 9 - 15), each in constant time. Hence, as the recursive cosntruction of the tree adds only a non dominant additive cost, the computational time of the algortihm is upper bounded by:

$$3^L \cdot (|A| + L) \leq (2^2)^L \cdot (|A| + L) \in O(|R|^2 \cdot \max\{|A|, |C|\}), \tag{3}$$

where C is the set of conditions (and $|C| = L$), R the set of rules (and $|R| = 2^L$), and A is the set of actions.

3 Optimizing Connected Components Labeling

A Connected Components Labeling algorithm assigns a unique identifier (an integer value, namely *label*) to every connected component of the image, in order to give the possibility to refer to it in the next processing steps. Usually, labeling algorithms deal with binary images.

The majority of images are stored in raster scan order, so the most common technique for connected components labeling applies sequential local operations in that order, as firstly introduced by Rosenfeld *et al.* [7]. This is classically performed in 3 steps, described in the following.

During the first step, each pixel label is evaluated locally by only looking at the labels of its already processed neighbors. When using 8-connectivity, these pixels belong to the scanning mask shown in Fig. 2(a). During the scanning procedure, pixels belonging to the same connected component can be assigned different (*provisional*) labels, which at some point will be marked as equivalent. In the second step, all the equivalent provisional labels must be merged into a single class. Modern algorithms process equivalences as soon as they are discovered (online equivalent labels resolution, as in [1]). Most of the recent optimizations techniques aim at increasing the efficiency of this step. Once the equivalences have been eventually solved, in the third step a second pass over the image is performed in order to assign to each foreground pixel the representative label of its equivalence class. Usually, the class representative is unique and it is set to be the minimum label value in the class.

The procedure of collecting labels and solving equivalences may be described by a *command execution metaphor*: the current and neighboring pixels provide a binary command word, interpreting foreground pixels as 1s and background pixels as 0s. A different action must be executed based on the command received. We may identify four different types of actions: *no action* is performed if the current pixel does not belong to the foreground, a *new label* is created when the neighborhood is only composed of background pixels, an *assign* action gives the current pixel the label of a neighbor when no conflict occurs (either only one pixel is foreground or all pixels share the same label), and finally a *merge* action is performed to solve an equivalence between two or more classes and a representative is assigned to the current pixel. The relation between the commands and the corresponding actions may be conveniently described by means of an *OR*-decision table.

As shown by Grana *et al.* [2], we can notice that, in algorithms with online equivalences resolution, already processed 8-connected foreground pixels cannot have different labels. This allows to remove merge operations between 8-connected pixels, substituting them with assignments of either of the involved pixels labels (many equivalent actions are possible, leading to the same result). It is also possible to enlarge the neighborhood exploration window, with the aim to speed up the connected components labeling process. The key idea comes from two very straightforward observations: when using 8-connection, the pixels of a 2×2 square are all connected to each other and a 2×2 square is the largest set of pixels in which this property holds. This implies that all foreground pixels in a the block will share the same label at the end of the computation. For this reason we propose to scan the image moving on a 2×2 pixel grid applying, instead of the classical neighborhood of Fig. 2(a), an extended mask of five 2×2 blocks, as shown in Fig. 2(b).

Employing all necessary pixels in the new mask of Fig. 2(b), we deal with $L = 16$ pixels (thus conditions), for a total amount of 2^{16} possible combinations. Grana *et al.* [2] employed the original Schumacher's algorithm, which required the conversion of the *OR*-decision table to a single entry decision table. This was performed with an heuristic technique, which led to producing a decision tree containing 210 nodes sparse over 14 levels, assuming all patterns occurred with the same probability and unitary cost for testing conditions. By using the algorithm proposed in this work, under the same assumptions, we obtain a much more compressed tree with 136 nodes sparse over 14 levels: the complexity in terms of levels is the same, but the code footprint is much lighter.

To push the algorithm performances to its limits, we further propose to add an occurrence probability for each pattern (p_r), which can be computed offline as a preprocessing stage on a subset of the dataset to be used, or the whole of it. The subset used for the probability computation obviously affects the algorithm performance (since we obtain a more or less optimal decision tree given those values), but the idea we want to carry out is that this optimization can be considered the upper bound of the performance we can obtain with this method.

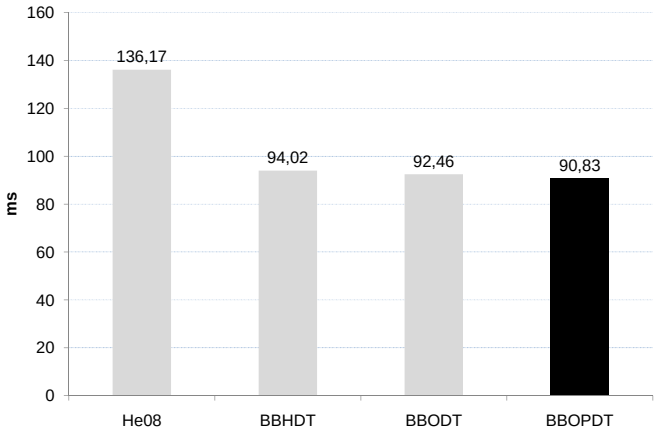


Fig. 3. The direct comparison between the He's approach (*He08*) with the three evolutions of block based decision tree approach, from the initial proposal with heuristic selection between alternative rules (*BBHDT*), further improved with the optimal decision tree generation (*BBOUDT*) and finally enhanced with a probabilistic weight of the rules (*BBOPDT*).

4 Experimental Results

In order to propose a valuable comparison with the state of the art, we used a dataset of Otsu-binarized versions of 615 high resolution document images. This dataset gives us the possibility to test the connected components labeling capabilities with very complex patterns at different sizes, with an average resolution of 10.4 megapixels and 35,359 labels, providing a challenging dataset which heavily stresses the algorithms. We performed a comparison between the following approaches:

- He's approach (*He07*), which highlights the benefits of the Union-Find algorithm for labels resolution implemented with the set of three arrays as in [3] and with the use of a decision tree to optimize the memory access with the mask of Fig. 2(a).
- The block based approach with decision tree generated with heuristic selection between alternatives as previously proposed by Grana *et al.* [2] (*BBHDT*)
- The block based approach with *optimal* decision tree generated with the procedure as proposed in this work, *assuming uniform distribution of patterns (*BBOUDT*)*
- The block based approach with *optimal* decision tree generated with the procedure weighted with the pattern probabilities (*BBOPDT*)

For each of these algorithms, the median time over five runs is kept in order to remove possible outliers due to other tasks performed by the operating system. All algorithms of course produced the same labeling on all images, and a unitary

cost is assumed for condition testing. The tests have been performed on a Intel Core 2 Duo E6420 processor, using a single core for the processing. The code is written in C++ and compiled on Windows 7 using Visual Studio 2008.

As reported in Fig. 3, we confirm the significant performance speedup presented by Grana *et al.* [2], which shows a gain of roughly 29% over the previous state-of-the-art approach of He *et al.* [4]. The optimal solution proposed in this work (BBODT) just slightly improves the performance of the algorithm. With the use of the probabilistic weight of the rules, in this case computed on the entire dataset, we can push the performance of the algorithm to its upper bound, showing that the optimal solution gains up to 3.4% of speedup over the original proposal. This last result, suggests that information about pattern occurrences should be used whenever available, or produced if possible.

5 Conclusions

In this paper we proposed a dynamic programming algorithm to generate an optimal decision tree from *OR*-decision tables. This decision tree represents the optimal arrangement of conditions to verify, and for this reason provides the fastest processing code to solve the neighborhood problem. The experimental section evidence how our approach can lead to faster results than other techniques proposed in literature. This suggests how this methodology can be considered a general modeling approach for many local image processing problems.

References

1. Di Stefano, L., Bulgarelli, A.: A simple and efficient connected components labeling algorithm. In: International Conference on Image Analysis and Processing. pp. 322–327 (1999)
2. Grana, C., Borghesani, D., Cucchiara, R.: Optimized Block-based Connected Components Labeling with Decision Trees. IEEE Transactions on Image Processing 19(6) (Jun 2010)
3. He, L., Chao, Y., Suzuki, K.: A linear-time two-scan labeling algorithm. In: International Conference on Image Processing. vol. 5, pp. 241–244 (2007)
4. He, L., Chao, Y., Suzuki, K., Wu, K.: Fast connected-component labeling. Pattern Recognition 42(9), 1977–1987 (Sep 2008)
5. Lew, A.: Optimal conversion of extended-entry decision tables with general cost criteria. Commun ACM 21(4), 269–279 (1978)
6. Reinwald, L.T., Soland, R.M.: Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time. J ACM 13(3), 339–358 (1966)
7. Rosenfeld, A., Pfaltz, J.L.: Sequential operations in digital picture processing. J ACM 13(4), 471–494 (1966)
8. Schumacher, H., Sevcik, K.C.: The Synthetic Approach to Decision Table Conversion. Commun ACM 19(6), 343–351 (1976)