

This is a pre print version of the following article:

Optimal Decision Trees for Local Image Processing Algorithms / Grana, Costantino; Montangero, Manuela; Borghesani, Daniele. - In: PATTERN RECOGNITION LETTERS. - ISSN 0167-8655. - STAMPA. - 33:16(2012), pp. 2302-2310. [10.1016/j.patrec.2012.08.015]

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

19/12/2025 04:19

Optimal Decision Trees for Local Image Processing Algorithms

Costantino Grana^{a,*}, Manuela Montangero^a, Daniele Borghesani^a

^a*Università degli Studi di Modena e Reggio Emilia, Dipartimento di Ingegneria dell'Informazione, Via Vignolese 905/b, 41125 Modena, Italy*

Abstract

In this paper we present a novel algorithm to synthesize an optimal decision tree from *OR*-decision tables, an extension of standard decision tables, complete with the formal proof of optimality and computational cost analysis. As many problems which require to recognize particular patterns can be modeled with this formalism, we select two common binary image processing algorithms, namely connected components labeling and thinning, to show how these can be represented with decision tables, and the benefits of their implementation as optimal decision trees in terms of reduced memory accesses. Experiments are reported, to show the computational time improvements over state of the art implementations.

Keywords: Decision trees; Decision tables; Connected components labeling; Thinning.

1. Introduction

Decision tables are a formalism used to describe the behavior of a system whose state can be represented by the outcome of testing certain conditions. Given a particular state, the system performs a set of actions. Each line of the table is a *rule*, which drives an action.

*Corresponding author. Tel. +39 059 205 6265, Fax. +39 059 205 6129.

Email addresses: `costantino.grana@unimore.it` (Costantino Grana), `manuela.montangero@unimore.it` (Manuela Montangero), `daniele.borghesani@unimore.it` (Daniele Borghesani)

6 A large class of image processing algorithms naturally leads to a decision
7 table specification, such as all those algorithms in which the output value for
8 each image pixel is obtained from the value of the pixel itself and of some of its
9 neighbors. We refer to this class as *local* algorithms. In particular for binary
10 images, we can model local algorithms by means of *decision tables*, in which the
11 pixels values are the conditions to be tested and the output is chosen by the
12 action corresponding to the conditions outcome.

13 Decision tables may be converted to decision trees in order to generate a
14 compact procedure to select the action to perform. Different decision trees for
15 the same decision table might lead to more or less tests to be performed, and
16 therefore to a higher or lower execution cost. The optimal decision tree is the
17 one that requires on average the minimum cost when deciding which action
18 execute [1].

19 In [2] we introduced a novel form of decision tables, namely *OR-Decision Ta-*
20 **bles**, which allow to include the representation of equivalent actions for a single
21 rule. An heuristic to derive a decision tree for such decision tables was given,
22 without guarantees on how good the derived tree was. In this paper, we further
23 develop that formalism by providing an exact dynamic programming algorithm
24 to derive optimal decision trees for such decision tables. The algorithm comes
25 with a formal proof of correctness and study of computational cost.

26 **2. Preliminaries and notation**

27 A decision table is a tabular form that presents a set of conditions which
28 must be tested and a list of corresponding actions to be performed: each row
29 corresponds to a particular outcome for the conditions and it is called *rule*, each
30 column corresponds to a particular set of actions to be performed. Different
31 rules might have different probability to occur and testing conditions might be
32 more or less expensive to test. We will call a decision table an *AND*-decision
33 table if *all* the actions in a row must be executed when the corresponding rule
34 occurs, instead we will call it an *OR*-decision table if *any* of the actions in a row

might be executed.

Schumacher *et al.* [1] proposed a bottom-up Dynamic Programming technique which guarantees to find the optimal decision tree given an expanded limited entry (binary) decision table, in which each row contains only one non-zero value. Lew [3] gives a Dynamic Programming approach for the case of extended entry and compressed *AND*-decision tables. In this paper, we extend Schumacher's approach to *OR*-decision tables. A preliminary version of this algorithm appeared in [4], where no proof of correctness was given.

In the following we will think of the set of rules as an L -dimensional Boolean space denoted by R , where $L \in \mathbb{N}$ is the given number of conditions. Testing conditions will be represented by position indexes of vectors in R , i.e. indexes in $[1 \dots L]$. Given any vector in R , a weight w_i is associated to each position index $i \in [1 \dots L]$, representing the cost of testing the condition in that particular position. Each vector in $r \in R$ has a given probability $p_r \geq 0$ to occur, such that $\sum_{r \in R} p_r = 1$.

We will call set $K \subseteq R$ a *k-cube* if it is a cube in $\{0, 1\}^L$ of dimension k , and it will be represented as a L -vector containing k dashes (–) and $L - k$ values 0's and 1's. The set of positions in which the vector contains dashes will be denoted as D_K . The occurrence probability of the k -cube K is the probability P_K of any element in K to occur, i.e. $P_K = \sum_{r \in K} p_r$. The set of all k -cubes, for each $k = 0, \dots, L$, will be denoted with \mathcal{K}_k .

Definition 1 (Extended Limited Entry *OR*-Decision Table). *Given a set of actions A , an extended limited entry *OR*-decision table is the description of a function $\mathcal{DT} : R \rightarrow 2^A \setminus \{\emptyset\}$, meaning that any action in $\mathcal{DT}(r)$ might be executed when $r \in R$ occurs.*

Given an *OR*-Decision Table \mathcal{DT} and a k -cube $K \in R$, set A_K denotes the actions (if any) that are common to all rules in K according to \mathcal{DT} ; i.e. $A_K = \cap_{r \in K} \mathcal{DT}(r)$ (might be an empty set).

Definition 2 (Decision Tree). *Given an *OR*-Decision Table \mathcal{DT} and a k -cube $K \subseteq R$, a Decision Tree for K , according to \mathcal{DT} , is a binary tree T with*

65 the following properties:

- 66 1. Each leaf ℓ corresponds to a k -cube, denoted by K_ℓ , that is a subset of K .
67 The cubes associated to the set of leaves of the tree are a partition of K .
68 Each leaf ℓ is associated to a non empty set of actions A_{K_ℓ} , associated
69 to cube K_ℓ by function \mathcal{DT} . Each internal node is labeled with an index
70 $i \in D_K$ (i.e. there is a dash at position i in the vector representation of
71 K) and is weighted by w_i . Left (resp. right) outgoing edges are labeled
72 with 0 (resp. 1).
- 73 2. Two distinct nodes on the same root-leaf path can not have the same label.
74 Root-leaf paths univocally identify, by means of nodes and edges labels, the
75 (vector representation of the) cubes associated to leaves: positions labeling
76 nodes on the path must be set to the value of the label on the corresponding
77 outgoing edges, the remaining positions are set to a dash.

78 When using decision tables to determine which action to execute, we need
79 to know the value assumed by exactly L conditions to identify the row of the
80 table that corresponds to the occurred rule. On the contrary, when we use a
81 decision tree (derived from the decision table) we only have to know the values
82 assumed by the conditions whose indexes label the root-leaf path leading to a
83 leaf associated to the cube that contains the occurred rule. This path might be
84 shorter than L , therefore using the tree we avoid to test the conditions that are
85 not on the root-leaf path. The sum of the weights of the missing conditions gives
86 an indication of the gain that we have, concerning that particular rule, in using
87 the tree instead of the table. On average, the gain in making a decision is given
88 by the sum of the gains given by rules in leaves, weighted by the probability
89 that the rules associated to leaves occur; for this reason, the gain of a tree is a
90 measure of the weights of the conditions that, on the average, we do not have
91 to test in order to decide which actions to take when rules occur.

92 **Definition 3 (Gain of a Decision Tree).** *Given a k -cube K and a decision*

93 tree T for K , the gain of T is defined in the following way:

$$\text{gain}(T) = \sum_{\ell \in \mathcal{L}} \left(P_{K_\ell} \sum_{i \in D_{K_\ell}} w_i \right), \quad (1)$$

94 where \mathcal{L} is the set of leaves of the tree, $D_{K_\ell} \subseteq D_K \subseteq [1 \dots L]$ is the set of position
 95 in which cube K_ℓ have dashes and the w_i s are their corresponding weights. An
 96 Optimal Decision Tree for k -cube K is a decision tree for the cube with maximum
 97 gain (might not be unique).

98 **Observation 1.** Given the definition of gain, we observe that:

- 99 1. If $P_K = 0$ for cube K , any decision tree for K has gain equal to zero as no
 100 element of the cube will ever occur. Moreover, a single leaf is the smallest
 101 possible tree representation of such a cube.
- 102 2. If a tree is a leaf ℓ , the gain of a leaf is well defined, as the summation in
 103 Eq. 1 has exactly one term, and $K = K_\ell$.
- 104 3. If a leaf ℓ corresponds to a 0-cube K_ℓ (meaning that all conditions must be
 105 tested), then the summation over indexes in D_{K_ℓ} is empty (being $|D_{K_\ell}| =$
 106 0) and the gain of the leaf is zero.
- 107 4. If a leaf has probability zero to occur, the gain is zero again. This makes
 108 sense, as there is no possible gain coming from rules that will never occur.

109 3. Optimal Decision Tree Generation from OR-Decision Tables

110 In order to derive a decision tree for a k -cube K it is possible to recursively
 111 proceed in the following way: select an index $j \in D_K$ (i.e. that is set to a dash)
 112 and make the root of the tree a node labeled with index j . Partition the cube K
 113 into two cubes $K_{j,0}$ and $K_{j,1}$ such that dash in position j is set to zero in $K_{j,0}$
 114 and to one in $K_{j,1}$. Recursively build decision trees for the two cubes of the
 115 partition, then make them the left and right children of the root, respectively.
 116 Recursion stops when the set of actions associated to a cube is non empty (i.e.
 117 $A_K \neq \{\emptyset\}$).

118 The gain of the obtained tree is strongly affected by the order used to select
119 the index that determines the cube partition. A *tree-compatible* partition is
120 a partition of cube K done according to an index j in D_K , in which index
121 j distinguishes between $K_{j,0}$ and $K_{j,1}$. There are k distinct tree-compatible
122 partition for any k -cube K , one for each different index in D_K . Moreover, each
123 subcube of the partition has dashes in the same positions given by set $D_K \setminus \{j\}$.
124 All rules of one subcube have condition in position j set to zero, while those in
125 the other subcube have that condition set to one.

126 **Proposition 1.** *Given a k -cube K and any tree-compatible partition $\{K_{j,0}, K_{j,1}\}$
127 for K we have*

$$P_K = P_{K_{j,0}} + P_{K_{j,1}} \quad \text{and} \quad A_K = A_{K_{j,0}} \cap A_{K_{j,1}}. \quad (2)$$

128 **PROOF.** The proof follows directly from the fact that $\{K_{j,0}, K_{j,1}\}$ is a partition
129 of K and from definitions of P_K and A_K . □

130 Observe that not all cube partitions are suitable for decision tree construc-
131 tion, only tree-compatible ones are. Consider, for example, cube $K = \{00, 01, 10, 11\}$
132 and the non tree-compatible partition $K' = \{00\}, K'' = \{01, 10, 11\}$. As-
133 sume that the intersection of actions associated to the cubes is empty (i.e.
134 $A_{K'} \cap A_{K''} = \{\emptyset\}$). Hence, the decision tree must have at least one internal
135 node. Assume we label the node with index $i = 1$. To satisfy decision trees
136 properties, rules of K' are to be placed in the subtree reached by following the
137 outgoing arc labeled with zero, while rules of K'' should be placed in the subtree
138 reached by following the outgoing arc labeled with one. But this is not possible
139 as rule $01 \in K''$ would be misplaced (it should be reached by following the out-
140 going arc labeled with one). Analogously, assume we label the node with index
141 $i = 2$, then rules of K' belong to the subtrees reached by following the outgoing
142 arc labeled with zero to satisfy decision trees property, and hence rules in K''
143 are to be placed in the subtree reached by following the outgoing arc labeled
144 with one. Again, this is impossible, as rule $10 \in K''$ is misplaced.

145 3.1. Dynamic Programming Algorithm

146 An optimal decision tree can be computed using a generalization of the
 147 Dynamic Programming strategy introduced by Schumacher *et al.* [1]: starting
 148 from 0-cubes and for increasing dimension of cubes, the algorithm computes the
 149 gain of all possible trees for all cubes and keeps track only of the ones having
 150 maximum gain. The pseudo-code is given in Algorithm 1.

151 To prove the algorithm correctness we first concentrate on leaves, than we
 152 move forward to trees with internal nodes.

153 **Lemma 1.** *Given an OR-Decision Table \mathcal{DT} and a k -cube K (for some $0 \leq$
 154 $k \leq L$), let A_K be the set of actions associated by \mathcal{DT} to cube K . If $P_K \neq 0$ and
 155 $A_K \neq \{\emptyset\}$, then the optimal decision tree for K is unique and it is composed of
 156 only one node (a leaf).*

157 **PROOF.** Assume, by contradiction, that there exist an optimal decision tree T
 158 for K with more than one node and such that $gain(T) = OPT$ is optimal.
 159 Then, there must exist two sibling leaves ℓ_0 and ℓ_1 such that:

- 160 1. $P_{\ell_0} > 0$ or $P_{\ell_1} > 0$ (if such a pair does not exist, then it must be $P_K = 0$,
 161 contradiction);
- 162 2. dashes of their corresponding cubes are in positions in set $D \subseteq D_K$ (being
 163 siblings, the set of positions is the same) such that $|D| = |D_K| - 1$;
- 164 3. their parent is node v , labeled with i , for some $1 \leq i \leq L$ and $i \notin D$;
- 165 4. $A_{\ell_0} \cap A_{\ell_1} \supseteq A_K \neq \{\emptyset\}$.

166 Build a new decision tree T' for K by replacing node v in T with a new leaf ℓ
 167 corresponding to the cube $K_{\ell_0} \cup K_{\ell_1}$, and associate set of actions $A_{\ell_0} \cap A_{\ell_1} \neq \{\emptyset\}$.
 168 The set of leaves of the new tree T' is given by $((\mathcal{L} \setminus (\ell_0 \cup \ell_1)) \cup \{\ell\})$ and the

Algorithm 1 MGDT - Maximum Gain Decision Tree for OR-Decision Tables

```

1: for  $K \in R$  do                                      $\triangleright$  Initialization of 0-cubes in  $R \in \mathcal{K}_0$ 
2:    $Gain_K^* \leftarrow 0$ 
3:    $A_K \leftarrow \mathcal{DT}(K)$   $\triangleright$  the set of actions associated to rule  $K$  by the OR-decision
     table
4:    $P_K \leftarrow p_K$   $\triangleright$  the occurrence probability of rule  $K$ 
5: end for
6: for  $n \in [1, L]$  do  $\triangleright$  for all possible cube dimensions  $> 0$ 
7:   for  $K \in \mathcal{K}_n$  do  $\triangleright$  for all possible cubes with  $n$  dashes
      $\triangleright$  compute current cube probability and set of actions by means of a
     tree-compatible partition
8:      $P_K \leftarrow P_{K_{j,0}} + P_{K_{j,1}}$   $\triangleright$  where  $j$  is any index in  $D_K$ 
9:      $A_K \leftarrow A_{K_{j,0}} \cap A_{K_{j,1}}$ 
10:    if  $P_K = 0$  then
11:       $Gain_K^* \leftarrow 0$ 
12:    else
13:      if  $A_K \neq \emptyset$  then
14:         $Gain_K^* \leftarrow w_j P_K + Gain_{K_{j,0}}^* + Gain_{K_{j,1}}^*$ 
15:      else  $\triangleright$  compute gains obtained by tree-compatible partitions, one at the
        time
16:        for  $i \in D_K$  do  $\triangleright$  for all positions set to a dash
17:           $Gain_K(i) \leftarrow Gain_{K_{i,0}}^* + Gain_{K_{i,1}}^*$ 
18:        end for
19:         $i_K^* \leftarrow \arg \max_{i \in D_K} Gain_K(i)$   $\triangleright$  keep the best gain and its index
20:         $Gain_K^* \leftarrow Gain_K(i_K^*)$ 
21:      end if
22:    end if
23:  end for
24: end for
25: BUILDTREE( $R$ )  $\triangleright$  recursively build tree on entire set of rules  $R \in \mathcal{K}_L$ 

26: procedure BUILDTREE( $K$ )
27:   if  $P_K = 0$  OR  $A_K \neq \emptyset$  then
      $\triangleright$  create leaf corresponding to cube  $K$  and associated to set of actions  $A_K$ 
28:     CREATELEAF( $A_K$ )
29:   else
      $\triangleright$  recursively build trees on subcubes given by tree-compatible partition
     distinguished by index  $i_K^*$ 
30:      $left \leftarrow \text{BUILDTREE}(K_{i_K^*,0})$ 
31:      $right \leftarrow \text{BUILDTREE}(K_{i_K^*,1})$ 
      $\triangleright$  create internal node labeled by index  $i_K^*$ , with subtrees build by recursive calls
32:     CREATENODE( $i_K^*, left, right$ )
33:   end if
34: end procedure

```

gain of T' might be computed in the following way:

$$\begin{aligned}
\text{gain}(T') &= \text{gain}(T) - [\text{gain}(\ell_0) + \text{gain}(\ell_1)] + \text{gain}(\ell) \\
&= \text{OPT} - \left[P_{\ell_0} \sum_{j \in D} w_j + P_{\ell_1} \sum_{j \in D} w_j \right] + \\
&\quad + P_{\ell} \sum_{j \in D \cup \{i\}} w_j \\
&= \text{OPT} + P_{\ell} w_i > \text{OPT},
\end{aligned} \tag{3}$$

as $P_{\ell} = P_{\ell_0} + P_{\ell_1} > 0$ and $w_i > 0$. Contradiction, T was supposed to have maximum gain. \square

Lemma 2. *Given an OR-Decision Table \mathcal{DT} and a k -cube K (for some $0 \leq k \leq L$), let A_K be the set of actions associated by \mathcal{DT} to cube K . If $P_K \neq 0$ and $A_K \neq \{\emptyset\}$, then algorithm MGDT associates to cube K a Gain_K^* such that*

$$\text{Gain}_K^* = P_K \sum_{i \in D_K} w_i. \tag{4}$$

PROOF. Proof is by induction on cube dimension. *Base case:* For 0-cubes we have (line 2) $\text{Gain}_K^* = 0 = P_K \sum_{i \in D_K} w_i$, as $D_K = \{\emptyset\}$. *Inductive hypothesis:* assume they are true for cubes such that $P_K \neq 0$ and $A_K \neq \{\emptyset\}$, having dimension up to $k-1$. *Inductive step:* Consider k -cube K such that $k > 0$, $P_K \neq 0$ and $A_K \neq \{\emptyset\}$. Then algorithm MGDT computes Gain_K^* according to line 14. Observe that, for any $j \in D_K$, the tree-compatible partition $\{K_{j,0}, K_{j,1}\}$ has the following properties: (1) $K_{j,0}$ and $K_{j,1}$ are $(k-1)$ -cubes; (2) $P_{K_{j,0}} + P_{K_{j,1}} = P_K$ and $\max\{P_{K_{j,0}}, P_{K_{j,1}}\} > 0$; (3) $A_{K_{j,0}}, A_{K_{j,1}} \neq \{\emptyset\}$ and (4) $D_{K_{j,0}} = D_{K_{j,1}} = D_K \setminus \{j\}$.

Suppose at first that $P_{K_{j,0}}, P_{K_{j,1}} > 0$, hence, inductive hypothesis applies to both $K_{j,0}$ and $K_{j,1}$ and

$$\begin{aligned}
Gain_K^* &= w_j P_K + Gain_{K_{j,0}}^* + Gain_{K_{j,1}}^* \quad (\text{line 14}) \\
&\quad \text{using the inductive hypothesis} \\
&= w_j P_K + P_{K_{j,0}} \sum_{i \in D_K \setminus \{j\}} w_i + P_{K_{j,1}} \sum_{i \in D_K \setminus \{j\}} w_i \\
&= P_K \sum_{i \in D_K} w_i.
\end{aligned}$$

186 Without loss of generality, suppose now that $P_{K_{j,0}} = 0$ and $P_{K_{j,1}} > 0$, then
187 inductive hypothesis applies only to $K_{j,1}$, $P_K = P_{K_{j,1}}$ and $Gain_{K_{j,0}}^* = 0$ (lines
188 10-11). We have

$$\begin{aligned}
Gain_K^* &= w_j P_K + Gain_{K_{j,1}}^* \quad (\text{line 14}) \\
&\quad \text{using the inductive hypothesis} \\
&= w_j P_K + P_K \sum_{i \in D_K \setminus \{j\}} w_i \\
&= P_K \sum_{i \in D_K} w_i.
\end{aligned}$$

189 □

190 **Corollary 1.** *If $P_K = 0$ or $A_K \neq \{\emptyset\}$, procedure BUILD TREE(K) computes an*
191 *optimal decision tree for K with only one leaf.*

192 **PROOF.** If $P_K = 0$, the algorithm associates to K a gain equal to zero (lines
193 10-11) and builds a tree that is a single leaf (line 28), optimal by definition and
194 observation 1.1.

195 If $A_K \neq \{\emptyset\}$ and $P_K \neq 0$, then by Lemma 1 the optimal tree must be a
196 leaf. The algorithm builds a tree that is a single leaf (line 28) to which it is
197 associated the gain of Equation (4) that is the definition of gain in the case in
198 which the tree is a leaf. □

Lemma 3. *Given an OR-Decision Table \mathcal{DT} and a k -cube K such that $P \neq 0$ and $A_K = 0$, let T be a decision tree for K of height $h \geq 1$ and let T_0 and T_1 be the subtrees of T . The gain of the tree might be recursively computed in the following way:*

$$\text{gain}(T) = \text{gain}(T_0) + \text{gain}(T_1).$$

199 PROOF. Let \mathcal{L} (resp. $\mathcal{L}_0, \mathcal{L}_1$) be the set of leaves of T (resp. T_0, T_1). We have
 200 that $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$, regardless from the fact that T_0 or T_1 are leaves or proper
 201 subtrees. We have

$$\begin{aligned} & \text{gain}(T_0) + \text{gain}(T_1) \\ &= \sum_{\ell \in \mathcal{L}_0} \left(P_{K_\ell} \sum_{j \in D_\ell} w_j \right) + \sum_{\ell \in \mathcal{L}_1} \left(P_{K_\ell} \sum_{j \in D_\ell} w_j \right) \\ &= \sum_{\ell \in \{\mathcal{L}_0 \cup \mathcal{L}_1\}} \left(P_{K_\ell} \sum_{j \in D_\ell} w_j \right) = \text{gain}(T). \end{aligned}$$

202 □

203 **Corollary 2.** *The maximum gain achievable by a decision tree for K is*

$$\max_{i \in D_K} (\text{gain}(K_{i,0}) + \text{gain}(K_{i,1})). \quad (5)$$

204 **Corollary 3.** *If $P_K \neq 0$ and $A_K = \{\emptyset\}$, procedure BUILDTREE(K) computes*
 205 *the optimal decision tree for K .*

206 Finally, we can conclude that

207 **Theorem 1.** *Given an expanded limited entry OR-Decision Table $\mathcal{DT} : \{0, 1\}^L \rightarrow$*
 208 *$2^A \setminus \{\emptyset\}$, algorithm MGDT computes an optimal decision tree.*

209 3.2. Computational time

210 The algorithm considers 3^L cubes, one for all possible words of length L on
 211 the three letter alphabet $\{0, 1, -\}$ (for cycles in lines 6 and 7). In the worst

case, for cube K of dimension n it computes: (1) the intersection of the actions associated to the cubes in one tree-compatible partition (line 9); this task can be accomplished, in the worst case, in time linear with the number of actions. (2) n gains, one for each index in D_K (lines 16 - 18), each in constant time.

The final recursive procedure for tree construction adds, in the worst case (in which a complete binary tree is constructed) an $O(2^L)$ term. Hence, the computational time of the algorithm is upper bounded by:

$$3^L \cdot (L + |A|) + 2^L \in O(3^L \cdot \max\{L, |A|\}). \quad (6)$$

3.3. About different types of decision tables

In literature other decision tables have been studied, representing functions having different domain or co-domain and different meaning.

Decision tables considered in [1] are description of functions $\mathcal{DT} : R \rightarrow A$, meaning that exactly one action to execute when rules occur. Therefore, these are a special case of the *OR*-decision tables considered in this paper (as $A \subset 2^A$) and our algorithm can be applied to those decision tables as well. In this case, however, the intersection of the set of actions can be accomplished in $O(1)$ computational time, leading to a tighter upper bound of the total computational running time, i.e. $O(3^L \cdot L)$.

AND-decision tables describe functions $\mathcal{DT} : R \rightarrow 2^A \setminus \{\emptyset\}$, meaning that *all* actions in $\mathcal{DT}(r)$ *must* be executed when rule r occurs, contrarily to what happens with *OR*-decision tables in which *any* action might be executed. Nevertheless, our algorithm might be applied also in this case with a simple pre-processing of the decision table: build a new set of *composed-actions* $\mathcal{A} = \{\mathcal{DT}(r) | r \in R\}$ and consider the *OR*-decision table that associates to rule r the composed-action $\mathcal{DT}(r)$. In this case, the worst case computational running time is upper-bounded by $O(2^L \cdot 2^{|A|} + 3^L \cdot L)$, where the first term comes from the table pre-processing (once this is done, intersections of the set of actions might be accomplished in $O(1)$ also in this case).

Compressed *OR*-Decision tables $\mathcal{DT} : \cup_{i \in [0..L]} \mathcal{K}_i \rightarrow 2^A \setminus \{\emptyset\}$ assign a set

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

273 As shown in [6], we can notice that, in algorithms with online equivalences
 274 resolution, already processed 8-connected foreground pixels cannot have dif-
 275 ferent labels. This allows to remove merge operations between these pixels,
 276 substituting them with assignments of either of the involved pixels labels. Ex-
 277 tending the same considerations throughout the whole rule set, we obtain the
 278 decision table of Fig. 2. Most of the *merge* operations are avoided, obtaining
 279 an *OR*-decision table with multiple alternatives between *assign* operations, and
 280 only in a single case between *merge* operations.

281 When using 8-connection, the pixels of a 2×2 square are all connected to
 282 each other and a 2×2 square is the largest set of pixels in which this property
 283 holds. This implies that all foreground pixels in a the block will share the same
 284 label. For this reason, scanning the image moving on a 2×2 pixel grid has the
 285 advantage to allow the labeling of four pixels at the same time.

286 Employing all necessary pixels in the enlarged neighborhood, we deal with
 287 $L = 16$ pixels(thus conditions), for a total amount of 2^{16} possible combinations.
 288 Using the approach described in [2] leads to producing a decision tree containing
 289 210 nodes sparse over 14 levels, assuming all patterns occurred with the same
 290 probability and unitary cost for testing conditions. Instead, by using the algo-

							assign				merge	
x	p	q	r	s	no action	new label	x = p	x = q	x = r	x = s	x = p+r	x = r+s
0	-	-	-	-	1	1						
1	1	0	0	0			1					
1	1	0	1	0				1				
1	0	0	1	0					1			
1	0	0	0	1						1		
1	1	1	1	0			1	1				
1	1	1	0	1							1	
1	1	0	0	1			1			1		
1	0	1	1	0				1	1			
1	0	1	0	1				1		1		
1	0	0	1	1								1
1	1	1	1	1			1	1	1			
1	1	1	0	1			1	1		1		
1	1	0	1	1							1	1
1	0	1	1	1			1	1	1	1		

Figure 2: The resulting *OR*-decision table for labeling

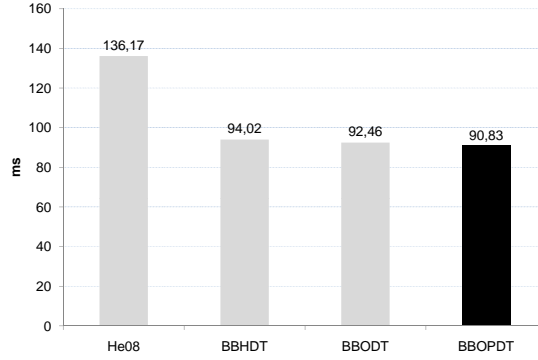


Figure 3: The direct comparison between the He’s approach (*He08*) with the three evolutions of block based decision tree approach, from the initial proposal with heuristic selection between alternative rules (*BBHDT*), further improved with the optimal decision tree generation (*BBOUDT*) and finally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

291 rithm proposed in this work, under the same assumptions, we obtain a much
 292 more compressed tree with 136 nodes sparse over 14 levels: the complexity in
 293 terms of levels is the same, but the code footprint is much lighter. Moreover, the
 294 resulting tree is proven to be the optimal one (Fig. 4). To push the algorithm
 295 performances to its limits, it is possible to add an occurrence probability for
 296 each pattern (p_r), which can be computed off-line as a preprocessing stage on a
 297 reference dataset.

298 To test the performance of the optimal decision tree, we used a dataset of
 299 Otsu-binarized versions of 615 high resolution page images of the Holy Bible of

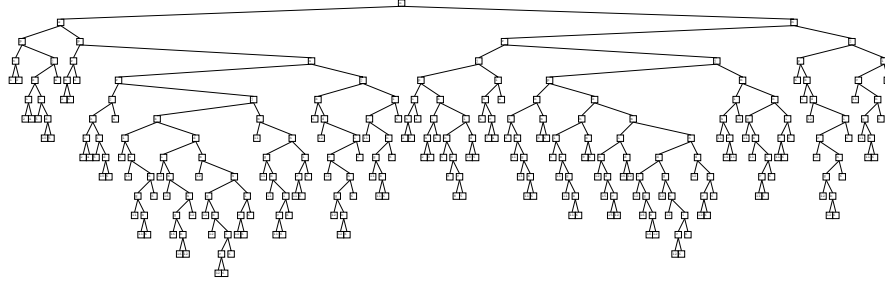


Figure 4: Optimal decision tree for *BBOUDT* method.

Borso d'Este, one of the most important Renaissance illuminated manuscript, composed by Gothic text, pictures and floral decorations. 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 35000 labels, providing a challenging dataset which heavily stresses the algorithms.

We performed a comparison between the following approaches:

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

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 of the BBHDT, which shows a gain of roughly 29% over the previous state-of-the-art approach of He *et al.*. 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.

4.2. Image Thinning

Thinning is a fundamental algorithm, often used in many computer vision tasks, such as document images understanding and OCR. A lot of algorithms have been detailed in literature to solve the problem, both in sequential or parallel fashion (according to the classification proposed by Lam *et al.* [7]).

One the most famous algorithms was proposed by Zhang and Suen [8]. The algorithm (ZS) consists in a two subiterations procedure in which a foreground pixel is removed if a set of conditions is satisfied. Starting from the current pixel P_1 , the neighboring pixels are enumerated in clockwise order:

P_9	P_2	P_3
P_8	P_1	P_4
P_7	P_6	P_5

Let $k = 0$ during the first subiteration and $k = 1$ during the second one. Pixel P_1 should be removed if the following conditions are true:

- a. $2 \leq B(P_1) \leq 6$
- b. $A(P_1) = 1$
- c. $P_2 * P_4 * P_6 = 0$ if $k = 0$

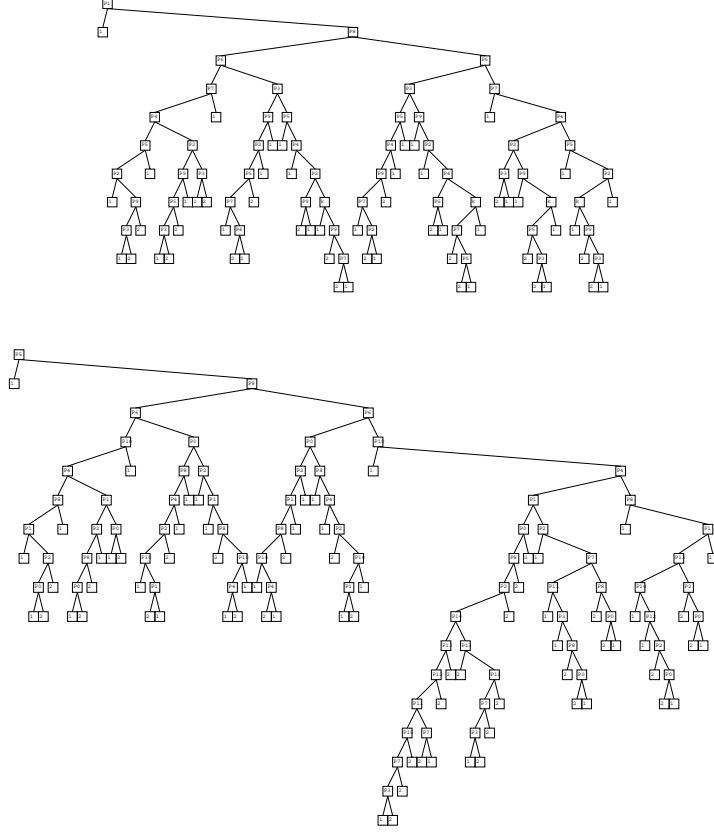


Figure 5: Decision trees for Zhang and Suen and Holt et al. thinning algorithms. The pixels in the 4×4 neighborhood are numbered in row major ordering, with current pixel being P_5 .

347 c'. $P_2 * P_4 * P_8 = 0$ if $k = 1$

348 d. $P_4 * P_6 * P_8 = 0$ if $k = 0$

349 d'. $P_2 * P_6 * P_8 = 0$ if $k = 1$

350 where $A(P_1)$ is the number of 01 patterns in clockwise order and $B(P_1)$ is the
 351 number of non zero neighbors of P_1 .

352 Holt *et al.* [9] algorithm (HSCP) is built on the ZS algorithm by defining
 353 an *edge* function $E(P)$ which returns true if, browsing the neighborhood in
 354 clockwise order, there are one or more 00 patterns, one or more 11 patterns and
 355 exactly one 01 pattern. The algorithm thus has a single type of iteration which
 356 removes a foreground pixel if the following conditions are true:

- 357 1. $E(P_1) = 1$
- 358 2. $E(P_4) * P_2 * P_6 = 0$
- 359 3. $E(P_6) * P_8 * P_4 = 0$
- 360 4. $E(P_4) * E(P_5) * E(P_6) = 0$

361 It should be noted that the edge function requires checking all neighbors of the
 362 analyzed pixel, thus the window used by the HSCP algorithm has a size of 4×4 .
 363 This algorithm reduces the number of iterations required, but the need to access
 364 more pixels makes it slower when implemented on sequential machines [10]

365 These thinning techniques can be modeled as decision tables in which the
 366 conditions are given by the fact that a neighboring pixel belongs to the fore-
 367 ground, and the only two possible actions are removing the current pixel or not.
 368 The ZS algorithm has also another condition, that is the value of subiteration
 369 index k . This results in a 9 conditions decision table for the ZS algorithm (512
 370 rules) and 16 conditions (the pixels of a 4×4 window) for HSCP algorithm
 371 (65536 rules). We ran the dynamic programming algorithm obtaining the two
 372 optimal decision trees shown in Fig. 5. We ignored patterns probabilities in this
 373 test. These trees represent the best access order for the neighborhood of each
 374 pixel. The leaves of the trees are the two actions: 1 means “do nothing”, while
 375 2 means “remove”. The left branch should be taken if the pixel referred in a
 376 node is background, otherwise the algorithm should follow the right one.

377 We compared the original ZS and HSCP with their version based on optimal
 378 decision trees. The procedures were used to thin a set of binary document im-
 379 ages, composed by 6105 high resolution scans of books taken from the Gutenberg
 380 Project [11], with an average amount of 1.3 millions of pixels. This is a typical
 381 application of document analysis and character recognition where thinning is a
 382 commonly employed preprocessing step.

383 The results of the comparison are reported in Table 1. The use of the decision
 384 trees significantly improves the performance of both ZS and HSCP algorithms.
 385 A second important result is that on average HSCP, despite being slower than
 386 ZS on sequential machines, becomes the fastest approach when the memory

Table 1: Comparison of the different thinning strategies and algorithms

	Average ms	fastest
ZS	1633	0%
ZS+Tree	1495	9%
HSCP	2493	0%
HSCP+Tree	1371	91%

access is optimized with our proposal. In fact in 91% of the cases, it turns out to be the fastest solution, mainly because the overall cost of an iteration is strongly reduced, thus the low number of iterations becomes the key factor in its success. With respect to the original ZS technique, the tree based version is around 10% faster, while HSCP is improved of around a 45%. This is supported by the observation that the larger the window, the higher the saving can be. HSCP+Tree is around 20% faster than the original ZS approach.

5. Conclusions

In this paper we presented a general modeling approach for local image processing problems, such as connected components labeling and thinning, by means of decision tables and decision trees. In particular, we leverage on *OR*-decision tables to formalize the situation in which multiple alternative actions could be performed, and proposed an algorithm to generate an optimal decision tree from the decision table with a formal proof of optimality. The experimental section evidence how our approach can lead to faster results than other techniques proposed in literature, and more importantly suggests how this methodology can be successfully applied to a lot of similar problems.

References

- [1] H. Schumacher, K. C. Sevcik, The Synthetic Approach to Decision Table Conversion, Commun ACM 19 (1976) 343–351.
- [2] C. Grana, D. Borghesani, R. Cucchiara, Optimized Block-based Connected

- 408 Components Labeling with Decision Trees, IEEE Transactions on Image
409 Processing 19 (2010).
- 410 [3] A. Lew, Optimal conversion of extended-entry decision tables with general
411 cost criteria, Commun ACM 21 (1978) 269–279.
- 412 [4] C. Grana, M. Montanero, D. Borghesani, R. Cucchiara, Optimal decision
413 trees generation from or-decision tables, in: Image Analysis and Processing
414 - ICIAP 2011, volume 6978, Ravenna, Italy, pp. 443–452.
- 415 [5] A. Rosenfeld, J. L. Pfaltz, Sequential operations in digital picture process-
416 ing, J ACM 13 (1966) 471–494.
- 417 [6] K. Wu, E. Otoo, A. Shoshani, Optimizing connected component labeling
418 algorithms, in: SPIE Conference on Medical Imaging, volume 5747, pp.
419 1965–1976.
- 420 [7] L. Lam, S.-W. Lee, C. Y. Suen, Thinning Methodologies—A Comprehen-
421 sive Survey, IEEE T Pattern Anal 14 (1992) 869–885.
- 422 [8] T. Y. Zhang, C. Y. Suen, A Fast Parallel Algorithm for Thinning Digital
423 Patterns, Commun ACM 27 (1984) 236–239.
- 424 [9] C. M. Holt, A. Stewart, M. Clint, R. H. Perrott, An Improved Parallel
425 Thinning Algorithm, Commun ACM 30 (1987) 156–160.
- 426 [10] R. W. Hall, Fast Parallel Thinning Algorithms: Parallel Speed and Con-
427 nectivity Preservation, Commun ACM 32 (1989) 124–131.
- 428 [11] Project Gutenberg, Project Gutenberg, <http://www.gutenberg.org>, 2010.