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)

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

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

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

2. Preliminaries and notation

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 row corresponds to a particular outcome for the conditions and it is called *rule*, each column corresponds to a particular set of actions to be performed. Different rules might have different probability to occur and testing conditions might be more o less expensive to test. 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

35 might be executed.

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

43 In the following we will think of the set of rules as an L-dimensional Boolean
44 space denoted by R, where $L \in N$ is the given number of conditions. Testing
45 conditions will be represented by position indexes of vectors in R, i.e. indexes in
46 $[1 \dots L]$. Given any vector in R, a weight w_i is associated to each position index
47 $i \in [1 \dots L]$, representing the cost of testing the condition in that particular
48 position. Each vector in $r \in R$ has a given probability $p_r \geq 0$ to occur, such
49 that $\sum_{r \in R} p_r = 1$.

50 We will call set $K \subseteq R$ a *k-cube* if it is a cube in $\{0,1\}^L$ of dimension k, and
51 it will be represented as a L-vector containing k dashes ($-$) and $L-k$ values
52 0's and 1's. The set of positions in which the vector contains dashes will be
53 denoted as D_K. The occurrence probability of the k-cube K is the probability
54 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,
55 for each $k = 0, \dots, L$, will be denoted with $\mathcal{K}_k$.

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

60 Given an *OR*-Decision Table $\mathcal{DT}$ and a k-cube $K \in R$, set A_K denotes
61 the actions (if any) that are common to all rules in K according to $\mathcal{DT}$; i.e.
62 $A_K = \cap_{r \in K} \mathcal{DT}(r)$ (might be an empty set) .

63 **Definition 2 (Decision Tree).** *Given an* OR-*Decision Table $\mathcal{DT}$ and a k-*
64 *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 form 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.

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

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

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

92 *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*
93 *in which cube K_ℓ have dashes and the w_is are their corresponding weights. An*
94 Optimal Decision Tree *for k-cube K is a decision tree for the cube with maximum*
95 *gain (might not be unique).*

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

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

107 3. Optimal Decision Tree Generation from *OR*-Decision Tables

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

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

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

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

Proof. The proof follows directly from the fact that $\{K_{j,0}, K_{j,1}\}$ is a partition of K and from definitions of P_K and A_k. □

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

141 *3.1. Dynamic Programming Algorithm*

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

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

149 **Lemma 1.** *Given an* OR*-Decision Table* $\mathcal{DT}$ *and a* k*-cube* K *(for some* $0 \leq$
150 $k \leq L$*), let* A_K *be the set of actions associated by* $\mathcal{DT}$ *to cube* K*. If* $P_K \neq 0$ *and*
151 $A_K \neq \{\emptyset\}$*, then the optimal decision tree for* K *is unique and it is composed of*
152 *only one node (a leaf).*

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

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

Build a new decision tree T' for K by replacing node v in T with a new leaf ℓ
corresponding to the cube $K_{\ell_0} \cup K_{\ell_1}$, and associate set of actions $A_{\ell_0} \cap A_{\ell_1} \neq \{\emptyset\}$.
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 ∈ R do                                   ▷ Initialization of 0-cubes in R ∈ 𝒦_0
2:     Gain*_K ← 0
3:     A_K ← 𝒟𝒯(K)    ▷ the set of actions associated to rule K by the OR-decision table
4:     P_K ← p_K                                   ▷ the occurrence probability of rule K
5: end for
6: for n ∈ [1, L] do                               ▷ for all possible cube dimensions > 0
7:     for K ∈ 𝒦_n do                              ▷ for all possible cubes with n dashes
           ▷ compute current cube probability and set of actions by means of a tree-compatible partition
8:         P_K ← P_{K_{j,0}} + P_{K_{j,1}}          ▷ where j is any index in D_K
9:         A_K ← A_{K_{j,0}} ∩ A_{K_{j,1}}
10:        if P_K = 0 then
11:            Gain*_K ← 0
12:        else
13:            if A_K ≠ ∅ then
14:                Gain*_K ← w_j P_K + Gain*_{K_{j,0}} + Gain*_{K_{j,1}}
15:            else▷ compute gains obtained by tree-compatible partitions, one at the time
16:                for i ∈ D_K do                   ▷ for all positions set to a dash
17:                    Gain_K(i) ← Gain*_{K_{i,0}} + Gain*_{K_{i,1}}
18:                end for
                                                   ▷ keep the best gain and its index
19:                i*_K ← arg max_{i∈D_K} Gain_K(i)
20:                Gain*_K ← Gain_K(i*_K)
21:            end if
22:        end if
23:    end for
24: end for
25: BuildTree(R)              ▷ recursively build tree on entire set of rules R ∈ 𝒦_L

26: procedure BuildTree(K)
27:     if P_K = 0 or A_K ≠ ∅ then
            ▷ create leaf corresponding to cube K and associated to set of actions A_K
28:         CreateLeaf(A_K)
29:     else
            ▷ recursively build trees on subcubes given by tree-compatible partition distinguished by index i*_K
30:         left ← BuildTree(K_{i*_K,0})
31:         right ← BuildTree(K_{i*_K,1})
        ▷ 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:

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

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

164 **Lemma 2.** *Given an* OR-*Decision Table* $\mathcal{DT}$ *and a* k*-cube* K *(for some* $0 \leq$
165 $k \leq L$*), let* A_K *be the set of actions associated by* $\mathcal{DT}$ *to cube* K*. If* $P_K \neq 0$
166 *and* $A_K \neq \{\emptyset\}$*, then algorithm* MGDT *associates to cube* K *a* $Gain_K^*$ *such that*

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

167 Proof. Proof is by induction on cube dimension. *Base case:* For 0-cubes we
168 have (line 2) $Gain_K^* = 0 = P_K \sum_{i \in D_K} w_i$, as $D_K = \{\emptyset\}$. *Inductive hypothesis:*
169 assume they are true for cubes such that $P_K \neq 0$ and $A_K \neq \{\emptyset\}$, having
170 dimension up to $k-1$. *Inductive step:* Consider k-cube K such that $k > 0$, $P_K \neq$
171 0 and $A_K \neq \{\emptyset\}$. Then algorithm MGDT computes $Gain_K^*$ according to line 14.
172 Observe that, for any $j \in D_K$, the tree-compatible partition $\{K_{j,0}, K_{j,1}\}$ has the
173 following properties: (1) $K_{j,0}$ and $K_{j,1}$ are $(k-1)$-cubes; (2) $P_{K_{j,0}} + P_{K_{j,1}} = P_K$
174 and $\max\{P_{K_{j,0}}, P_{K_{j,1}}\} > 0$; (3) $A_{K_{j,0}}, A_{K_{j,1}} \neq \{\emptyset\}$ and (4) $D_{K_{j,0}} = D_{K_{j,1}} =$
175 $D_K \setminus \{j\}$.

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

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

178 Without loss of generality, suppose now that $P_{K_{j,0}} = 0$ and $P_{K_{j,1}} > 0$, then
179 inductive hypothesis applies only to $K_{j,1}$, $P_K = P_{K_{j,1}}$ and $Gain^*_{K_{j,0}} = 0$ (lines
180 10-11). We have

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

181 □

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

184 Proof. If $P_K = 0$, the algorithm associates to K a gain equal to zero (lines
185 10-11) and builds a tree that is a single leaf (line 28), optimal by definition and
186 observation 1.1.

187 If $A_K \neq \{\emptyset\}$ and $P_K \neq 0$, then by Lemma 1 the optimal tree must be a
188 leaf. The algorithm builds a tree that is a single leaf (line 28) to which it is
189 associated the gain of Equation (4) that is the definition of gain in the case in
190 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:*

$$gain(T) = gain(T_0) + gain(T_1).$$

191 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
192 that $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$, regardless form the fact that T_0 or T_1 are leaves or proper
193 subtrees. We have

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

194 □

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

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

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

197 Finally, we can conclude that

198 **Theorem 1.** *Given an expanded limited entry* OR*-Decision Table $\mathcal{DT} : \{0,1\}^L \rightarrow$*
199 *$2^A \setminus \{\emptyset\}$, algorithm* MGDT *computes an optimal decision tree.*

200 ### *3.2. Computational time*

201 The algorithm considers 3^L cubes, one for all possible words of length L on
202 the three letter alphabet $\{0, 1, -\}$ (for cycles in lines 6 and 7). In the worst
203 case, for cube K of dimension n it computes: (1) the intersection of the actions
204 associated to the cubes in one tree-compatible partition (line 9); this task can
205 be accomplished, in the worst case, in time linear with the number of actions.
206 (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:

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

207 ### *3.3. About different types of decision tables*

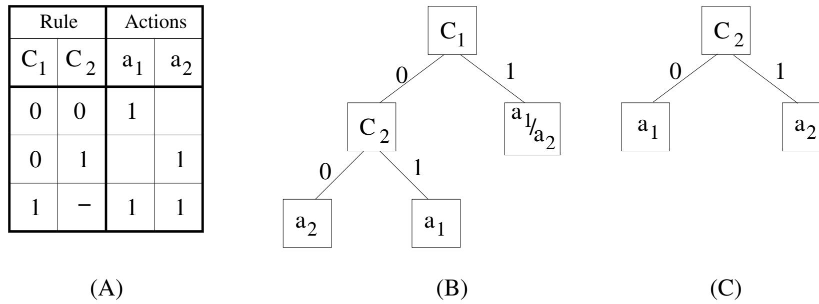
208 In literature other decision tables have been studied, representing functions
209 having different domain or co-domain and different meaning.

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

217 *AND*-decision tables describe functions $\mathcal{DT} : R \rightarrow 2^A \setminus \{\emptyset\}$, meaning that *all*
218 actions in $\mathcal{DT}(r)$ *must* be executed when rule r occurs, contrarily to what hap-
219 pens with *OR*-decision tables in which *any* action might be executed. Neverthe-
220 less, our algorithm might be applied also in this case with a simple pre-processing
221 of the decision table: build a new set of *composed-actions* $\mathcal{A} = \{\mathcal{DT}(r) | r \in R\}$
222 and consider the *OR*-decision table that associates to rule r the composed-action

$\mathcal{DT}(r)$. In 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 $\mathcal{DT} : \cup_{i \in [0..L]} \mathcal{K}_i \to 2^A \setminus \{\emptyset\}$ assign a set of actions to cubes of rules. One might think that the algorithm might be used also in this case, by just making a leaf associated to all the rules in the cube that corresponds to a compressed rule. In Figure 1 we give a very simple example showing that, this approach, does not lead to the optimal decision tree. Hence, to derive a decision tree starting from a compressed table, we first have to expand the table (and might get a new table with size exponential in the size of the original one) or use a different approach.



Rule		Actions	
C_1	C_2	a_1	a_2
0	0	1	
0	1		1
1	–	1	1

Figure 1: Example showing that algorithm MGDT can not be applied to compressed *OR*-Decision tables without expanding the table. (A) Compressed *OR*-Decision table with two conditions C_1 and C_2, and two actions a_1 and a_2. We have $w_i = 1$ for all conditions, and $p_i = 1/4$ for all rules. (B) and (C) Decision trees for the table in (A): labels of internal nodes correspond to the conditions to be tested. Labels of leaves correspond to actions to take (if more than one action is present, this means that any can be taken). Labels on edges represent conditions testing outcome. (B) Tree build without splitting compressed rule $1-$. This tree has gain $1/2$. (C) Tree build by splitting rule $1-$ having gain 1, greater that (B).

4. Decision Tables Applied to Image Processing Problems

In this section we show how the described approach can be effectively applied to two common image processing tasks: connected components labeling and thinning. The former requires the use of *OR*-decision tables, while the latter only requires two mutually exclusive actions, thus implicitly leads to a single

entry decision table. Anyway, both can be improved by the application of the proposed technique.

4.1. Connected components labeling

Labeling algorithms take care of the assignment of 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. This is classically performed in 3 steps [5]: provisional labels assignment and collection of label equivalences, equivalences resolution, and final label assignment.

During the first step, each pixel label is evaluated 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). As mentioned before, during the scanning procedure, the same connected component can be assigned different (*provisional*) labels, so all algorithms adopt some mechanism to keep track of the possible equivalences.

In the second step, all the provisional labels must be segregated into disjoint sets, or disjoint equivalence classes. As soon as an unprocessed equivalence is considered (online equivalent labels resolution, as in [6]), a "merging" between classes is needed, that is some operation which allows to mark as equivalent all labels involved. Most of the recent optimizations introduced in modern connected components labeling techniques aim at increasing the efficiency of this step (*Union-Find* algorithm [7]).

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 is set to be the minimum label value in the class.

In the recent literature, a set of works enclosed the main innovations in the field of connected components labeling. In 2005, Wu *et al.* [8] defined an interesting optimization to reduce the number of labels by exploiting a decision tree to minimize the number of neighboring pixels to be visited in order to evaluate the label of the current pixel. Authors indeed observed that in a 8-

270 connected components neighborhood, among all the neighboring pixels, often
271 only one of them is sufficient to determine the label of the current pixel. In
272 the same paper, authors proposed also a strategy to improve the Union-Find
273 algorithm by means of an array-based data structure. In 2007, He *et al.* [9]
274 proposed another fast approach in the form of a two scan algorithm. The
275 data structure used to manage the label resolution is implemented using three
276 arrays in order to link the sets of equivalent classes without the use of pointers.
277 Adopting this data structure, He *et al.* [10] proposed a decision tree to optimize
278 the neighborhood exploration applying merging only when needed.

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

292 As shown in [8], we can notice that, in algorithms with online equivalences
293 resolution, already processed 8-connected foreground pixels cannot have dif-
294 ferent labels. This allows to remove merge operations between these pixels,
295 substituting them with equivalent actions like assignments of either of the in-
296 volved pixels labels. Extending the same considerations throughout the whole
297 rule set, we can transform the original naïve decision table of Fig. 2(a) into
298 the OR-decision table of Fig. 2(c), in which most of the *merge* operations are
299 avoided and multiple alternatives between *assign* operations are available.

300 When using 8-connection, the pixels of a 2×2 square are all connected to

p	q	r
s	x	

(a)

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

(b)

							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	0	0	0	0		**1**						
1	1	0	0	0			**1**					
1	0	1	0	0				**1**				
1	0	0	1	0					**1**			
1	0	0	0	1						**1**		
1	1	1	0	0			*1*	**1**				
1	1	0	1	0							**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	0			*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	1	1	1	1			*1*	**1**	*1*	*1*		

(c)

Figure 2: The *OR*-decision table obtained by applying the nieghborhood information in 8-connection. We get rid of most of the merge operations by alternatively using more lightweight assign operations. An heuristic or an exhaustive search can be used to select the most convenient action among the alternatives, here represented by bold 1s.

301 each other and a 2×2 square is the largest set of pixels in which this property
302 holds. This implies that all foreground pixels in a the block will share the same
303 label. For this reason, scanning the image moving on a 2×2 pixel grid has the
304 advantage to allow the labeling of four pixels at the same time.

305 Employing all necessary pixels in the enlarged neighborhood, we deal with
306 $L = 16$ pixels(thus conditions), for a total amount of 2^{16} possible combinations.
307 Using the approach described in [2] leads to producing a decision tree containing
308 210 nodes sparse over 14 levels, assuming all patterns occurred with the same
309 probability and unitary cost for testing conditions. Instead, by using the algo-
310 rithm proposed in this work, under the same assumptions, we obtain a much
311 more compressed tree with 136 nodes sparse over 14 levels: the complexity in
312 terms of levels is the same, but the code footprint is much lighter. Moreover, the
313 resulting tree is proven to be the optimal one (Fig. 4). To push the algorithm
314 performances to its limits, it is possible to add an occurrence probability for
315 each pattern (p_r), which can be computed off-line as a preprocessing stage on a

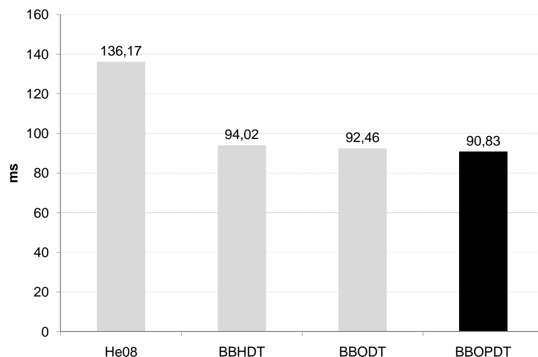


Figure 3: The direct comparison between the He's approach (*He08*) with the three evolutions of block based decision tree approach, from the initial proposal with heuristic selection between alternative rules (*BBHDT*), further improved with the optimal decision tree generation (*BBOUDT*) and finally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

316 reference dataset.

317 To test the performance of the optimal decision tree, we used a dataset of
318 Otsu-binarized versions of 615 high resolution page images of the Holy Bible of
319 Borso d'Este, one of the most important Renaissance illuminated manuscript,
320 composed by Gothic text, pictures and floral decorations. This dataset gives us
321 the possibility to test the connected components labeling capabilities with very
322 complex patterns at different sizes, with an average resolution of 10.4 megapixels
323 and 35000 labels, providing a challenging dataset which heavily stresses the
324 algorithms.

325 We performed a comparison between the following approaches:

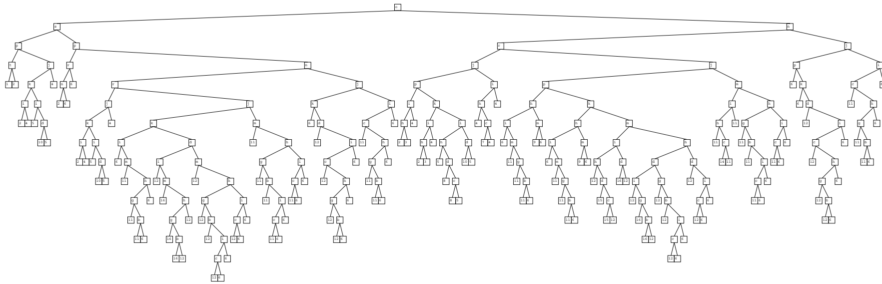
- 326 He *et al.* [10] approach (*He08*), which highlights the benefits of the Union-
327 Find algorithm for labels resolution and the use of a decision tree to op-
328 timize the memory access.

- 329 The block based approach with decision tree generated with heuristic se-
330 lection between alternatives as previously proposed in [2] (*BBHDT*)

- 331 The block based approach with *optimal* decision tree generated with the
332 procedure proposed in this work, *assuming uniform distribution of pat-*
333 *terns (*BBOUDT*)*

a	b	c	d	e	*f*
g	h	i	j	k	*l*
m	n	o	p		
q	r	s	t		

(a)



(b)

Figure 4: The extended mask used in *BBOUDT* method and the optimal decision tree obtained. Each node in the tree represents a pixel of the mask to check, each leaf represents a possible action to take, the left branch means having a background pixel while the right branch means having a foreground pixel (for more details, please refer to [2]). For example, reading the tree we can say that if o is background, s is foreground, p is background and r is foreground (thus checking 4 pixel over 16) is sufficient to take an action for all the foreground pixels of the current block (action *6* in Fig.17 of [2] corresponds to the assigment of the label of block S, its leftmost one.)

- 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

P9	P2	P3
P8	P1	P4
P7	P6	P5

Figure 5: Pixels in the 4×4 neighborhood are numbered in row major ordering, with current pixel being P_5.

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.

Source code and datasets used in this tests are publicly available online [11]. Both the machine and the datasets are the same used in [2].

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.* [12]).

One the most famous algorithms was proposed by Zhang and Suen [13]. 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 as shown in Fig. 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 $k = 0$

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

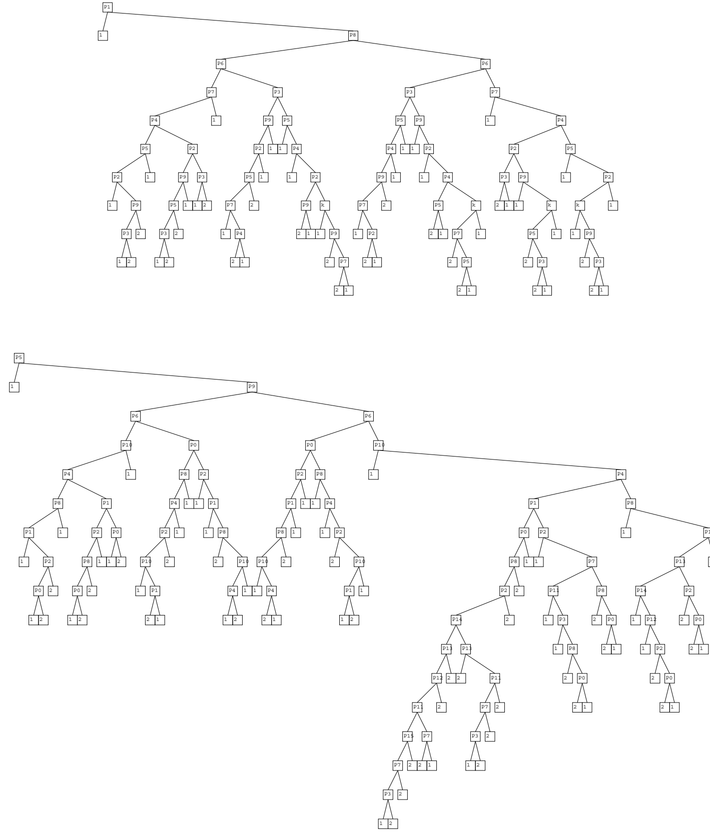


Figure 6: Decision trees for Zhang and Suen and Holt et al. thinning algorithms. As for labeling, nodes represent pixels to check, leafs represent actions to take, that in the case of labeling are limited to mantain or remove the current pixel. The left branch means having a background pixel while the right branch means having a foreground pixel.

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

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

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

Holt *et al.* [14] algorithm (HSCP) is built on the ZS algorithm by defining an *edge* function $E(P)$ which returns true if, browsing the neighborhood in clockwise order, there are one or more 00 patterns, one or more 11 patterns and exactly one 01 pattern. The algorithm thus has a single type of iteration which

377 removes a foreground pixel if the following conditions are true:

378 1. $E(P_1) = 1$

379 2. $E(P_4) * P_2 * P_6 = 0$

380 3. $E(P_6) * P_8 * P_4 = 0$

381 4. $E(P_4) * E(P_5) * E(P_6) = 0$

382 It should be noted that the edge function requires checking all neighbors of the
383 analyzed pixel, thus the window used by the HSCP algorithm has a size of 4×4.
384 This algorithm reduces the number of iterations required, but the need to access
385 more pixels makes it slower when implemented on sequential machines [15]

386 These thinning techniques can be modeled as decision tables in which the
387 conditions are given by the fact that a neighboring pixel belongs to the fore-
388 ground, and the only two possible actions are removing the current pixel or not.
389 The ZS algorithm has also another condition, that is the value of subiteration
390 index k. This results in a 9 conditions decision table for the ZS algorithm (512
391 rules) and 16 conditions (the pixels of a 4×4 window) for HSCP algorithm
392 (65536 rules). We ran the dynamic programming algorithm obtaining the two
393 optimal decision trees shown in Fig. 6. We ignored patterns probabilities in this
394 test. These trees represent the best access order for the neighborhood of each
395 pixel. The leaves of the trees are the two actions: 1 means "do nothing", while
396 2 means "remove". The left branch should be taken if the pixel referred in a
397 node is background, otherwise the algorithm should follow the right one.

398 We compared the original ZS and HSCP with their version based on optimal
399 decision trees. The procedures were used to thin a set of binary document im-
400 ages, composed by 6105 high resolution scans of books taken from the Gutenberg
401 Project [16], with an average amount of 1.3 millions of pixels. This is a typical
402 application of document analysis and character recognition where thinning is a
403 commonly employed preprocessing step.

404 The results of the comparison are reported in Table 1. The use of the decision
405 trees significantly improves the performance of both ZS and HSCP algorithms.
406 A second important result is that on average HSCP, despite being slower then

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%

ZS on sequential machines, becomes the fastest approach when the memory 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.

428 [2] C. Grana, D. Borghesani, R. Cucchiara, Optimized Block-based Connected
429 Components Labeling with Decision Trees, IEEE Transactions on Image
430 Processing 19 (2010).

431 [3] A. Lew, Optimal conversion of extended-entry decision tables with general
432 cost criteria, Commun ACM 21 (1978) 269–279.

433 [4] C. Grana, M. Montangero, D. Borghesani, R. Cucchiara, Optimal decision
434 trees generation from or-decision tables, in: Image Analysis and Processing
435 - ICIAP 2011, volume 6978, Ravenna, Italy, pp. 443–452.

436 [5] A. Rosenfeld, J. L. Pfaltz, Sequential operations in digital picture process-
437 ing, J ACM 13 (1966) 471–494.

438 [6] L. Di Stefano, A. Bulgarelli, A simple and efficient connected components
439 labeling algorithm, in: International Conference on Image Analysis and
440 Processing, pp. 322–327.

441 [7] Z. Galil, G. F. Italiano, Data structures and algorithms for disjoint set
442 union problems, ACM Computing Surveys 23 (1991) 319–344.

443 [8] K. Wu, E. Otoo, A. Shoshani, Optimizing connected component labeling
444 algorithms, in: SPIE Conference on Medical Imaging, volume 5747, pp.
445 1965–1976.

446 [9] L. He, Y. Chao, K. Suzuki, A linear-time two-scan labeling algorithm, in:
447 International Conference on Image Processing, volume 5, pp. 241–244.

448 [10] L. He, Y. Chao, K. Suzuki, K. Wu, Fast connected-component labeling,
449 Pattern Recognition 42 (2008) 1977–1987.

450 [11] labeling, ImageLab optimized block based con-
451 nected components labeling source code and datasets,
452 http://imagelab.ing.unimore.it/imagelab/labeling.asp, 2012.

453 [12] L. Lam, S.-W. Lee, C. Y. Suen, Thinning Methodologies—A Comprehen-
454 sive Survey, IEEE T Pattern Anal 14 (1992) 869–885.

455 [13] T. Y. Zhang, C. Y. Suen, A Fast Parallel Algorithm for Thinning Digital
456 Patterns, Commun ACM 27 (1984) 236–239.

457 [14] C. M. Holt, A. Stewart, M. Clint, R. H. Perrott, An Improved Parallel
458 Thinning Algorithm, Commun ACM 30 (1987) 156–160.

459 [15] R. W. Hall, Fast Parallel Thinning Algorithms: Parallel Speed and Con-
460 nectivity Preservation, Commun ACM 32 (1989) 124–131.

461 [16] Project Gutenberg, Project Gutenberg, http://www.gutenberg.org, 2010.