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 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 $L \in N$ 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 $\mathcal{DT} : R \to 2^A \setminus \{\emptyset\}$, 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* *the following properties:*

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

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

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

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

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

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

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

In order to derive a decision tree for a k-cube K it is possible to recursively proceed in the following way: select an index $j \in D_K$ (i.e. that is set to a dash) and make the root of the tree a node labeled with index j. Partition the cube K into two cubes $K_{j,0}$ and $K_{j,1}$ such that dash in position j is set to zero in $K_{j,0}$ and to one in $K_{j,1}$. 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 the set of actions associated to a cube is non empty (i.e. $A_K \neq \{\emptyset\}$).

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. $\square$

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.

3.1. Dynamic Programming Algorithm

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

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

Lemma 1. *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 the optimal decision tree for* K *is unique and it is composed of only one node (a leaf).*

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

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

for $K \in R$ **do** ▷ Initialization of 0-cubes in $R \in \mathcal{K}_0$
  $Gain^*_K \leftarrow 0$
  $A_K \leftarrow \mathcal{DT}(K)$ ▷ the set of actions associated to rule K by the OR-decision table
  $P_K \leftarrow p_K$ ▷ the occurrence probability of rule K
end for
for $n \in [1, L]$ **do** ▷ for all possible cube dimensions > 0
  for $K \in \mathcal{K}_n$ **do** ▷ for all possible cubes with n dashes
    ▷ compute current cube probability and set of actions by means of a tree-compatible partition
    $P_K \leftarrow P_{K_{j,0}} + P_{K_{j,1}}$ ▷ where j is any index in D_K
    $A_K \leftarrow A_{K_{j,0}} \cap A_{K_{j,1}}$
    if $P_K = 0$ **then**
      $Gain^*_K \leftarrow 0$
    else
      if $A_K \neq \emptyset$ **then**
        $Gain^*_K \leftarrow w_j P_K + Gain^*_{K_{j,0}} + Gain^*_{K_{j,1}}$
      else▷ compute gains obtained by tree-compatible partitions, one at the time
        for $i \in D_K$ **do** ▷ for all positions set to a dash
          $Gain_K(i) \leftarrow Gain^*_{K_{i,0}} + Gain^*_{K_{i,1}}$
        end for
        ▷ keep the best gain and its index
        $i^*_K \leftarrow \arg\max_{i \in D_K} Gain_K(i)$
        $Gain^*_K \leftarrow Gain_K(i^*_K)$
      end if
    end if
  end for
end for
BuildTree(R) ▷ recursively build tree on entire set of rules $R \in \mathcal{K}_L$

procedure BuildTree(K)
  if $P_K = 0$ or $A_K \neq \emptyset$ **then**
    ▷ create leaf corresponding to cube K and associated to set of actions A_K
    CreateLeaf(A_K)
  else
    ▷ recursively build trees on subcubes given by tree-compatible partition distinguished by index i^*_K
    $left \leftarrow$ BuildTree($K_{i^*_K,0}$)
    $right \leftarrow$ BuildTree($K_{i^*_K,1}$)
    ▷ create internal node labeled by index i^*_K, with subtrees build by recursive calls
    CreateNode($i^*_K, left, right$)
  end if
end procedure

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

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

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

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*

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

Proof. Proof is by induction on cube dimension. *Base case:* For 0-cubes we have (line 2) $Gain^*_K = 0 = P_K \sum_{i \in D_K} w_i$, 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

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

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

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

□

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

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

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

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

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

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

$\square$

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

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

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

Finally, we can conclude that

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

3.2. Computational time

The algorithm considers 3^L cubes, one for all possible words of length L on 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:

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

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 $\mathcal{DT} : R \to A$, 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 $\mathcal{DT} : R \to 2^A \setminus \{\emptyset\}$, 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 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.

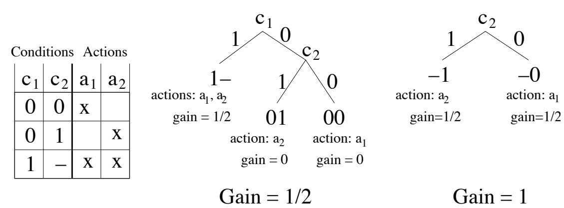

Figure 1: $|L| = |A| = 2$, $w_i = 1$ for all conditions, $p_i = 1/4$ for all rules, action a_1 associated to rule 01, actions $\{a_1, a_2\}$ to rules $1-$, action a_2 to rule 00. The tree build by taking $1-$ as a "block" has gain 1/2, if we split the block we get a greater gain of 1.

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.

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 taken 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 a decision table.

As shown in [6], 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 these pixels, substituting them with assignments of either of the involved pixels labels. Extending the same considerations throughout the whole rule set, we obtain the decision table of Fig. 2. Most of the *merge* operations are avoided, obtaining an *OR*-decision table with multiple alternatives between *assign* operations, and only in a single case between *merge* operations.

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. For this reason, scanning the image moving on a 2×2 pixel grid has the advantage to allow the labeling of four pixels at the same time.

Employing all necessary pixels in the enlarged neighborhood, we deal with $L = 16$ pixels(thus conditions), for a total amount of 2^{16} possible combinations. Using the approach described in [2] leads 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. 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	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*		

Figure 2: The resulting OR-decision table for labeling

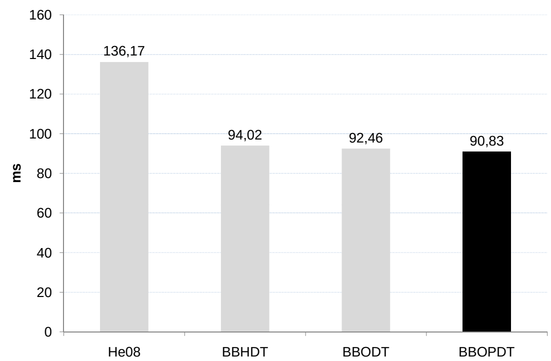

Figure 3: The direct comparison between the He's approach ($He08$) with the three evolutions of block based decision tree approach, from the initial proposal with heuristic selection between alternative rules ($BBHDT$), further improved with the optimal decision tree generation ($BBOUDT$) and finally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

rithm 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. Moreover, the resulting tree is proven to be the optimal one (Fig. 4). To push the algorithm performances to its limits, it is possible to add an occurrence probability for each pattern (p_r), which can be computed off-line as a preprocessing stage on a reference dataset.

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

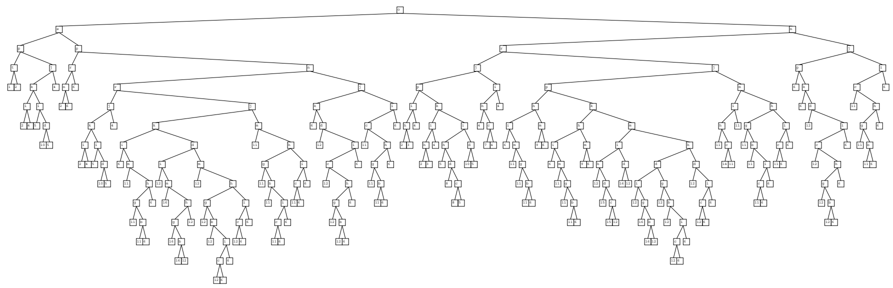

Figure 4: Optimal decision tree for $BBOUDT$ method.

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 $k = 0$

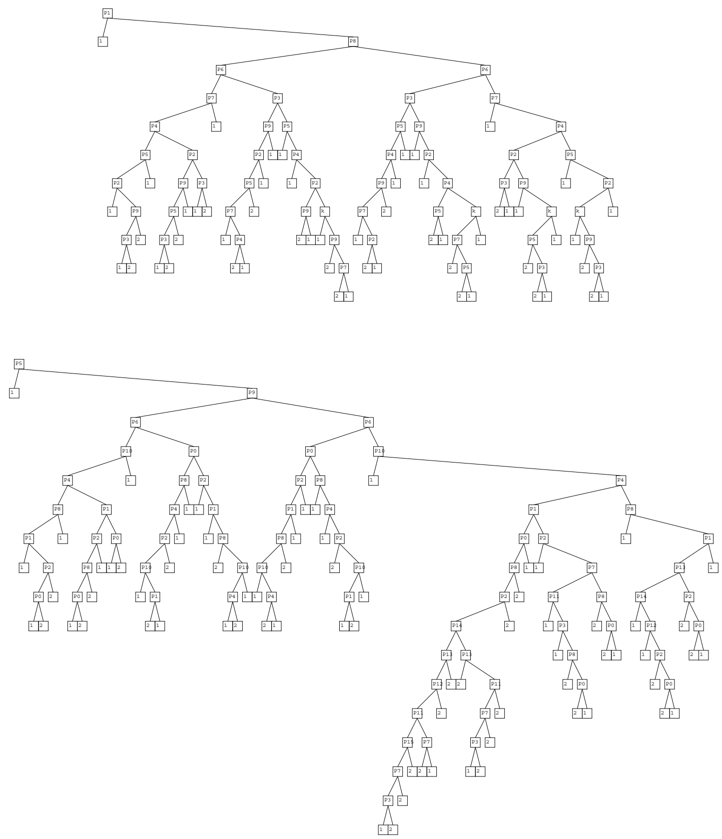

Figure 5: Decision trees for Zhang and Suen and Holt et al. thinning algorithms. The pixels in the 4×4 neighborhood are numbered in row major ordering, with current pixel being P_5.

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

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.* [9] 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 removes a foreground pixel if the following conditions are true:

1. $E(P_1) = 1$
2. $E(P_4) * P_2 * P_6 = 0$
3. $E(P_6) * P_8 * P_4 = 0$
4. $E(P_4) * E(P_5) * E(P_6) = 0$

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

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

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

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

Components Labeling with Decision Trees, IEEE Transactions on Image Processing 19 (2010).

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

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

[5] A. Rosenfeld, J. L. Pfaltz, Sequential operations in digital picture processing, J ACM 13 (1966) 471–494.

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

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

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

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

[10] R. W. Hall, Fast Parallel Thinning Algorithms: Parallel Speed and Connectivity Preservation, Commun ACM 32 (1989) 124–131.

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