Fast Binary Image Processing Using Binary Decision Diagrams

Luc Robert, Grégoire Malandain

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fast Binary Image Processing Using Binary Decision Diagrams

Luc Robert, Grégoire Malandain

N° 3001

Octobre 1996

THÈME 3

ISSN 0249-6399

Fast Binary Image Processing Using Binary Decision Diagrams

Luc Robert* , Grégoire Malandain**

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projets Robotvis, Épidaure

Rapport de recherche n˚3001 — Octobre 1996 — 24 pages

Abstract: Many classical image processing tasks can be realized as evaluations of a boolean function over subsets of an image. For instance, the simplicity test used in 3D thinning requires examining the 26 neighbors of each voxel and computing a single boolean function of these inputs. In this article, we show how Binary Decision Diagrams can be used to produce automatically very efficient and compact code for such functions. The total number of operations performed by a generated function is at most one test and one branching for each input value (e.g., in the case of 3D thinning, 26 tests and branchings). At each stage, the function is guaranteed to examine only the pertinent input data, i.e., the values which affect the result.

As an example, we consider the 2D and 3D simplicity tests in digital topology, and thinning processes. We produce functions much faster than our previously optimized implementations [18, 4], and than any other implementation we know of. In the case of 3D simplicity test, on average, at each voxel only 8.7 neighboring voxel values are examined.

Key-words: binary image processing, BDD, mathematical morphology, digital topology

(Résumé : tsvp)

* Email : Luc.Robert@sophia.inria.fr
** Email : Gregoire.Malandain@sophia.inria.fr

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Traitement Rapide d'Images Binaires à l'aide de Diagrammes de Décision Binaires

Résumé : Un certain nombre de techniques de traitement d'image reposent sur l'évaluation d'une fonction booléenne sur des parties de l'image. Par exemple, la caractérisation des points simples utilisée pour l'amincissement 3D requiert d'examiner les 26 voisins de chaque voxel, et de calculer une fonction booléenne de ces valeurs. Dans cet article, nous montrons comment utiliser les Diagrammes de Décision Binaires pour produire automatiquement du code très efficace et compact implémentant de telles fonctions. Le nombre total d'opérations effectuées par la fonction produite est au plus d'un test et un branchement par valeur d'entrée (c'est-à-dire, dans le cas de l'amincissement 3D, 26 test et branchements). À chaque étape, la fonction n'examine que les valeurs qui ont une influence sur le résultat.

Comme application, nous considérons la caractérisation des points simples en topologie discrète 2D et 3D, et plusieurs techniques d'amincissement 3D. Nous produisons des fonctions beaucoup plus efficaces que nos implémentations optimisées existantes [18, 4], et que toutes les autres implémentations que nous connaissons. Pour caractériser les points simples en 3D, il suffit en moyenne pour chaque voxel d'examiner 8.7 voisins.

Mots-clé : traitement d'images binaires, bdd, morphologie mathématique, topologie discrète

1 Introduction

Many classical image processing techniques proceed by analyzing at each element of the image (pixel, voxel) the values of its neighbors. Considering the particular case of binary images, a number of techniques for image processing have been developed in the domains of mathematical morphology [26, 27], digital topology [13], or for other specific tasks such as image edge linking [10, 22]. Each of these techniques relies on at least one function which, for a given pixel, analyzes the values of its neighbors. This function has to be evaluated many times during the process. In fact, a large fraction of the computational effort is devoted to evaluating this function, and the implementation of the function itself has a tremendous effect on the efficicency of the whole process. Several solutions have been proposed for efficient implementations so far.

First, it is sometimes possible to decompose the original function into more elementary functions, which can be evaluated very efficiently. For instance, in mathematical morphology, a 3x3x3 structuring element is *separable* and can be decomposed into the product of three structuring elements (3x1x1, 1x3x1 and 1x1x3).

When this is not possible, a very classical approach consists of building a lookup-table for the whole state space [15]. This approach yields code which evaluates in constant time: all the input variables are examined once in order to compute the address of the entry in the lookup-table. If N pixels are examined, the number of entries of the lookup table is 2^N. This quickly becomes a very large number as N increases. Large lookup tables are undesirable, for reasons of efficiency of memory access or insufficient memory in the host computer.

When lookup-tables are infeasible due to lack of space, efficient implementations can still be obtained by using appropriate algorithmic tools. For instance, in digital topology, graph theory can be applied to efficiently count the connected components in a small neighborhood [18].

Sometimes, a trade-off is chosen between space and time complexity. Algorithmic considerations are used to reduce the problem into smaller sub-problems, for instance by taking into account the symmetries of the problem. The sub-problems are then solved using lookup-tables or quadtrees [17, 15, 24, 12].

We propose another approach to the problem, which relies on the use of Binary Decision Diagrams (BDD's), as a convenient representation for boolean functions of boolean variables. First, we compute the BDD which represents the boolean func-

tion to be evaluated. This can be done either by formal boolean computation, or by "brute-force" processing. We then compile the BDD into efficient C code. The generated code has the following interesting properties: first, contrary to all the techniques mentioned above, the code generated using BDD's is guaranteed to examine only the pixels whose value affects the result. Second, for each examined pixel value, only performs one test, one branching, and sometimes one binary operation on a register.

In section 2, we give a brief description of BDD's (a detailed description and review can be found in [7]). Section 3 shows how we use them to generate efficient code for the computation of functions of boolean variables. In section 4 we show the results obtained when applying this method to the computation of simple points in 2D and 3D, and to thinning in 3D. We finally give some conclusions in section 5.

2 Binary Decision Diagrams

Binary Decision Diagrams are a very compact and efficient representation for symbolic manipulation of boolean functions. Up to now, they have been mostly used in digital-system design and finite-state system analysis [7]. Their concept was introduced by Lee [16] and Akers [1]. A BDD represents a boolean function $f(x_1, .., x_n)$ as a directed acyclic graph, each node of which corresponds to a test of a boolean variable x_i. Terminal nodes of the graph are the function values (0 or 1). For instance, in figure 1 we show two different BDD representations of the boolean function $f(x_1, x_2) = x_1 + x_2$ (the boolean *and* and *or* are respectively denoted by $.$ and $+$).

Bryant showed [7] that by imposing a fixed order on the variables and by sharing identical sub-graphs, it was possible to reduce drastically the size of the structure. The obtained representation, called a Reduced Ordered Binary Decision Diagrams (ROBDD), is canonical. In other words, if variable ordering is fixed and reduction rules are applied, two equivalent boolean functions are guaranteed to have the same BDD. In figure 2 we show the ROBDD for the function of figure 1.

To compact the ROBDD further, the concept of *complement edges* was introduced [5]. A type (say, positive or negative) is assigned to each edge. The fact that an edge is negative indicates that the boolean value computed by the pointed sub-graph has to be complemented when evaluating the function. This representation allows a formula and its negation to share the same graph. It is also canonical.

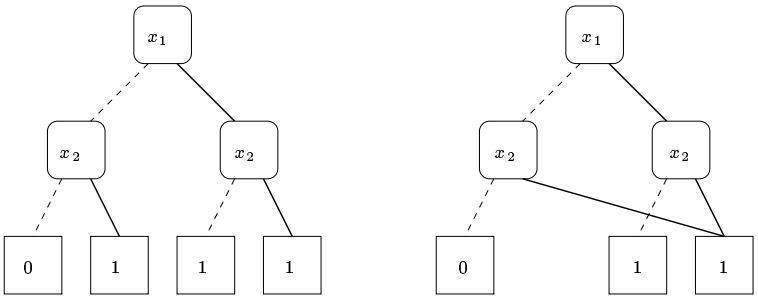

Figure 1: Two BDD representations of $f(x_1, x_2) = x_1 + x_2$. Each node represents a test over one variable. Solid lines represent the "then" branches, dotted lines the "else" branches. The representation on the left is the complete tree corresponding to the Boole-Shannon expansion of f, whereas the diagram on the right is more compact.

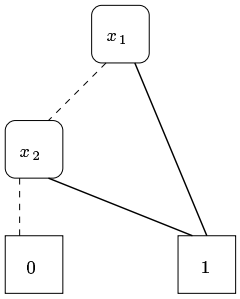

Figure 2: The ROBDD representation of $f(x_1, x_2) = x_1 + x_2$ for the ordering $x_1 < x_2$. It contains only 2 non-terminal nodes.

From now on, by BDD we will mean “Reduced Ordered BDD with complement edges”.

From the practical standpoint, there exist a number of packages for BDD manipulation. State-of-the-art BDD packages implement reduced ordered BDDs with complement edges, include efficient memory management packages, and allow dynamic reordering of the input variables during BDD creation to reduce the BDD size on the fly. They include programmatic interfaces for creating BDD functions, controlling variable ordering, and controlling memory management. For the experiments described in the remainder of this article, we used the package described in [6].

3 Generating fast code for a discrete function

In this section we will show how we can automatically produce an efficient implementation of a given discrete function, either specified by a formal description in terms of boolean operations, or induced from an implementation which may be inefficient. We first describe how we compute a BDD representing the function. Then, we show how this BDD can be translated into efficient C-code, based on methods used in digital system design [9, 8]. We first consider the case of a boolean function. Then, we consider the more general case of a discrete function (i.e., one which may take a discrete set of values).

3.1 Computing the BDD corresponding to a boolean function

Let us consider a boolean function f depending on the n boolean variables $x_1, .., x_n$, also represented by the n-vector $\mathbf{x} = (x_1, .., x_n)$. One instance of such a function, for example, would tell if a pixel of a two-dimensional image can be removed in a thinning operation, based on the values of its 8 neighbors.

3.1.1 Formal derivation

Many functions used in mathematical morphology have a direct expression in terms of boolean operations over pixel values [26]. For instance, in the case of erosion, a pixel with value 1 has to be set to 0 if and only if at least one of its neighbors (for

a given structuring element) is 0. So, the function for updating the pixel value is as follows:

$$f(\mathbf{x}) = \prod_{1 \leq i \leq n} x_i$$

where x_i are boolean variables representing the pixel values around the pixel of interest, in the domain defined by the structuring element.

In such cases, the BDD of the boolean function can be easily computed formally, by converting in a straightforward manner boolean operations into BDD manipulations.

3.1.2 Brute-force method

In a second case, we assume that no simple boolean description of the function is known, but we have access to one implementation of f. This is the case, for instance, for the simplicity test in 3D digital topology. Another example is a function for which we have access to object code but not to source code, or a function which uses a lookup-table of unknown structure.

Let us denote by $\mathbf{s} = (s_1, .., s_n) \in \{0,1\}^n$ a n-vector of binary values. Please note that x_1,..,x_n are binary variables, while $s_1, .., s_n$ is a collection of values that represent a point in $\{0,1\}^n$. Let S_f be the satisfying set of f, i.e.,

$$\mathbf{s} \in S_f \;\Leftrightarrow\; f(\mathbf{s}) = 1$$

we have the following property:

$$f(\mathbf{x}) = \sum_{\mathbf{s} \in S_f} \left(\prod_{1 \leq i \leq n} (s_i.x_i + \overline{s_i}.\overline{x_i}) \right) \quad (1)$$

Based on this property, we can compute the BDD of f incrementally, by scanning the whole range of $\mathbf{x}$, i.e., $\{0,1\}^n$. For each assignment $\mathbf{s}$ of the variables, we compute the output value of the function (0 or 1) using the available implementation. If the function evaluates to 1, we update the BDD of f using equation (1).

3.2 Compiling the BDD into C-code

Let us first assume that there are no complement edges in the BDD. Then, each non-terminal node of the BDD can be considered as a quadruple (x, L, L_l, L_r) where x

is the boolean variable tested in the node, and L, L_l, L_r are respectively labels assigned to the node and its left and right sons. The terminal nodes are assigned labels `L_TRUE` and `L_FALSE`.

For each non-terminal node (x, L, L_l, L_r) of the BDD, we can generate the following line of C-code:

`L:` `if (` x `) goto` L_l `; else goto` L_r`;`

The code corresponding to the terminal nodes is added (see example below). All the lines are grouped into a procedure which takes as input an array of boolean variables, and returns a boolean value. The code corresponding to the root node of the BDD has to be executed first in the procedure. For instance, the BDD represented in figure 2 could be translated into the following function:

```
BOOLEAN func(BOOLEAN x[])
{
  L_0:     if ( x[0] ) goto L_TRUE; else goto L_1;
  L_1:     if ( x[1] ) goto L_TRUE; else goto L_FALSE;
  L_FALSE: return BOOLEAN_0;
  L_TRUE:  return BOOLEAN_1;
}
```

Another way of proceeding consists of keeping the nested structure of the BDD in the generated code. For instance, if the BDD is a binary decision tree (cf. figure 1, first case), we can trivially generate the corresponding arborescence of tests. If not (i.e. simplification rules have been applied), we can still generate the tests corresponding to the tree obtained by depth-first traversal of the BDD. All the edges of the BDD which do not belong to the traversal tree (i.e. those which, during the traversal, lead to already-visited nodes) are converted into `goto` instructions.

The BDD of figure 2 would be then translated into:

```
BOOLEAN func(BOOLEAN x[])
{
  if ( x[0] )
       return BOOLEAN_1;
  else
       if ( x[1] )
            return BOOLEAN_1;
       else
            return BOOLEAN_0;
}
```

An easy optimization, actually provided by the ROBDD package, consists of removing the tests on the last variable.

```
BOOLEAN func(BOOLEAN x[])
{
  if ( x[0] )
       return BOOLEAN_1;
  else
       return x[1];
}
```

We can easily adapt the same translation schemes to the case of diagrams with complement edges. In this case, the resulting code has to keep track during the descent of the diagram of the traversed negative branches. For this purpose, we add a binary register R whose initial value is false, and whose status is flipped each time a negative edge is traversed. When reaching a node, the value of this register is true if and only if the returned value has to be negated. Thus, for positive edges, the generated lines of code are identical as above. In the case of a negative edge, a `R = !R;` instruction has to be executed before moving to the next test. An example of such code will be given in the next section.

Finding an efficient translation strategy is highly dependent on the optimization criterion (size of the code, computational time, etc) and on the machine on which the code is to be run. In [9], a method for finely adapting the generated code to different machine architectures is presented. Using such refinements could probably allow us to produce code adapted to a given machine or architecture, and, in turn, more

efficient. In the experiments presented in the remainder of the paper, we focus on one particular criterion: we try to minimize the number of accesses to image data. This is indeed one of the most penalizing operations for most architectures where a small piece of code manipulates a large amount of data.

3.3 The case of a discrete function

In this section we show how the same scheme can be applied to generate efficient code for discrete – not only boolean – function evaluation. Let us consider a discrete function depending on the vector of input boolean variables $\mathbf{x} = (x_1, .., x_n)$. We assume that there are at most 2^k output values. Thus, each value can be represented by an assignment of k *output* boolean variables, represented by the vector $\mathbf{o} = (o_1, .., o_k)$.

We represent f by the characteristic function ϕ of its graph. In other words,

$$\phi(x_1, .., x_n, o_1, .., o_k) = 1 \Leftrightarrow f(\mathbf{x}) = \mathbf{o}$$

3.3.1 Computing ϕ

Computation of the BDD of ϕ is analogous to what has been presented above: If we denote by $f_j(\mathbf{s})$ the value of the jth component of $f(\mathbf{s})$, we have

$$\phi(x_1, .., x_n, o_1, .., o_k) = \sum_{\mathbf{s} \in \{0,1\}^n} \left(\prod_{1 \leq i \leq n} (s_i.x_i + \overline{s_i}.\overline{x_i}). \prod_{1 \leq j \leq k} (f_j(\mathbf{s}).o_j + \overline{f_j(\mathbf{s})}.\overline{o_j}) \right) \quad (2)$$

We compute the BDD of ϕ incrementally, by scanning the whole range of $(x_1, .., x_n)$, i.e., $\{0, 1\}^n$. For each assignment $(s_1, .., s_n)$ of the variables, we compute the output value of the function using the available implementation, and update the BDD of ϕ using equation (2).

3.3.2 Generating C-code

Let us assume that the BDD variable ordering is $x_1, .., x_n, o_1, .., o_k$. Since all input variables appear before output variables, we obtain the configuration depicted in figure 3.3.2: For a given assignment of the input variables, we follow a path of the BDD, until we reach a node which does not correspond to an input variable. This

node necessarily corresponds to an output variable or a constant value. Because of the fact that f is a function, we know that

- this node corresponds to the first output variable o_1;
- there is one and only one path issued from this node and leading to 1.

Indeed, the contrary would mean that there exists an set of input values for which several assignments of the output variables are possible, implying that the graph of f is not functional.

Further more, due to the basic properties of ROBDD's, we also know that from every node corresponding to variable o_1, one of the two edges leads directly to 0. Indeed,

$$\forall (t_2, .., t_k) \in \{0,1\}^{k-1}, \phi(s_1, .., s_n, \overline{f_1(\mathbf{s})}, t_2, .., t_k) = 0$$

In other words, given that input data are $(s_1, .., s_n)$, if o_1 is set to $\overline{f_1(\mathbf{s})}$, then the value of ϕ does not depend on the last $k-1$ output variables.

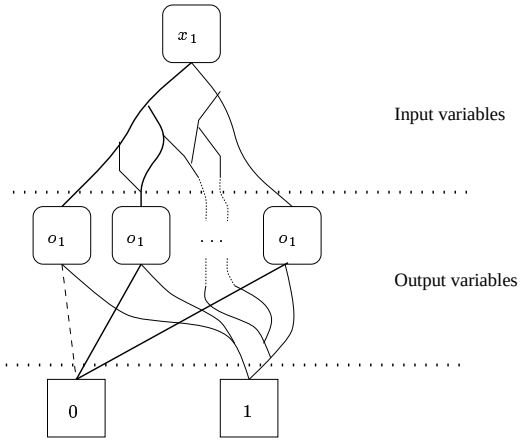

Figure 3: The ROBDD of the graph of a discrete function (see text).

4 Applications to binary image processing

We applied the code generation techniques presented above to several image processing problems. We present here some results obtained with the detection of simple

points in 2D and 3D, and the problem of thinning without shrinking in 3D. In each case we estimate the complexity of the generated code, in terms of the average number of tests on image values. Let us first recall some basic definitions from 2D and 3D digital topology [23, 13, 21, 28].

4.1 Basic definitions of digital topology

A 2D (respectively 3D) digital image Σ is a subset of $\mathbb{Z}^2$ (resp. $\mathbb{Z}^3$). A point $x \in \Sigma$ is defined by its coordinates (x_1, x_2) (resp. (x_1, x_2, x_3)). We consider two distances:

$$\begin{aligned} D_1(x,y) &= \sum_{i=1}^{i=2 \text{ (resp.3)}} \mid y_i - x_i \mid \qquad \text{and} \\ D_\infty(x,y) &= \max_{i=1\ldots2 \text{ (resp.3)}} \mid y_i - x_i \mid \end{aligned}$$

and the associated neighborhoods:

$$\begin{aligned} B_1^r(x) &= \{y \mid D_1(x,y) \leq r\} \qquad \text{and} \\ B_\infty^r(x) &= \{y \mid D_\infty(x,y) \leq r\} \end{aligned}$$

The common neighborhoods are:

in 2D	4-neighborhood: $N_4(x) = B_1^1(x)$
	8-neighborhood: $N_8(x) = B_\infty^1(x)$
in 3D	6-neighborhood: $N_6(x) = B_1^1(x)$
	26-neighborhood: $N_{26}(x) = B_\infty^1(x)$
	18-neighborhood: $N_{18}(x) = B_1^2(x) \cap B_\infty^1(x)$

We define $N_n^*(x) = N_n(x) \setminus \{x\}$. Two points x and y are said to be n-adjacent if $y \in N_n^*(x)$. We call the points of $N_6^*(x)$ the 6-neighbors of x.

A binary image Σ is defined by a binary partition of Σ into X and $\overline{X}$ such that $X \cap \overline{X} = \emptyset$ and $X \cup \overline{X} = \Sigma$. We call X the *object* and $\overline{X}$ the *background*.

A n-path between two points x and y of X (resp. $\overline{X}$) is a sequence $x_0, \ldots x_k$ of points such that $x_i \in X$ (resp. $\overline{X}$), $x_0 = x$, $x_k = y$, and x_{i-1} is n-adjacent to x_i for $i = 1 \ldots k$. A subset of X (resp. $\overline{X}$) is said to be n-connected if a n-path can be found between each pair of points of X (resp. $\overline{X}$). A n-connected component

of X (resp. $\overline{X}$) is a subset of X (resp. $\overline{X}$) which is n-connected and maximal for inclusion.

To preserve topological consistency, we have to consider different connectivities for the object and the background [13]. We respectively denote them by n and $\overline{n}$. We consider here that $(n, \overline{n}) = (8, 4)$ in 2D and $(n, \overline{n}) = (26, 6)$ or $(6, 26)$ in 3D.

4.2 Simple point

A simple point is a point whose removal does not change the topology of the binary image. The detection of such points is the keypoint of all thinning algorithms [13, 14], thus optimizing this detection is of high interest in computer vision and image processing. The detection and the characterization of 2D simple points has been already widely studied. Due to the simplicity of the two-dimensional problem, several very efficient implementations have been proposed. Because of new 3D imaging modalities (e.g., medical imaging), the study of the 3D case is becoming more and more important. It is also much more complex. Morgenthaler [20] proposed a first characterization of 3D simple points by using 4 local conditions. Tsao and Fu [29, 13] proposed a simplified form which requires 3 local conditions. Recently, characterizations with only 2 local conditions has been found [19, 4, 25].

This last characterization [4] is based on the calculation of two numbers of connected components (we consider that the object is 26-connected while the background is 6-connected):

$$(x \text{ is simple }) \Longleftrightarrow \Big(NCC_{26}[X \cap N^*_{26}(x)] = NCC_6[\overline{X} \cap N^*_{18}(x)] = 1\Big) \qquad (3)$$

$NCC_{26}[X \cap N^*_{26}(x)]$ denotes the number of 26-connected components (26-adjacent to x) of the object X in $N^*_{26}(x)$, while $NCC_6[\overline{X} \cap N^*_{18}(x)]$ denotes the number of 6-connected components (6-adjacent to x) of the background $\overline{X}$ in $N^*_{18}(x)$.

In the remainder of this section, we first present the results obtained in the simple case of detecting 2D simple points. The interest in this case is more to demonstrate the code generation process on a well-known, simple example. Then, we present the results obtained on the detection of 3D simple points, and the problem of thinning in 3D.

4.3 Simple points in 2D

Indexed from 0 to 7, the neighbors of a pixel are considered in the following order: NW, N, NE, W, E, SW, S, SE (i.e. from left to right, lines being considered from top to bottom). Figure 4 represents the initial – small, but not efficient – procedure used for BDD generation. The generated BDD is represented in figure 5. The C-code procedures automatically generated using the two different strategies presented in section 3.2 are displayed in figures 6 and 7. The generated code has a minimal size over all the potential variable orderings. Indeed, by computing all the $8!$ BDD's corresponding to all variable permutations, we checked that the variable order obtained by applying the *sifting* minimization algorithm actually corresponds to a global minimum for the number of nodes. Note that this exhaustive search was possible only because of the small number of variables.

In table 1, we compare the numbers of tests performed by the initial function and by the generated code. Both procedures were run on the 2^8 potential configurations of the input values. The table displays, for each number of tests, the corresponding number of configurations.

number of tests	4	5	6	7	8	9	12	average
initial function (fig. 4)	16	64	32	64	0	64	16	7.00
BDDs function (fig. 6)	144	64	32	8	8			4.72

Table 1: 2D simplicity test: comparison of the initial function with the generated code.

4.4 Detecting simple points in 3D

In [3], a boolean characterization of 3D simple points is given, which uses 4 boolean conditions based on local topological numbers.

A straightforward implementation of this characterization was then used to generate, using the "brute-force" approach (cf section 3.1.2), a BDD representing the function which returns TRUE if and only if the point is a simple point.

Computation took several hours on a *Sun Sparc 20*. As a result, we obtained a BDD with 1007 nodes. Using the *sifting* minimization algorithm, we reduced its size

```
int is_simple_2D_initial (const int *V) {
    int nb_cc = 0;
    if ( !V[1] ) { nb_cc ++; if ( !V[4] && !V[2] ) nb_cc --; }
    if ( !V[4] ) { nb_cc ++; if ( !V[6] && !V[7] ) nb_cc --; }
    if ( !V[6] ) { nb_cc ++; if ( !V[3] && !V[5] ) nb_cc --; }
    if ( !V[3] ) { nb_cc ++; if ( !V[1] && !V[0] ) nb_cc --; }

    if ( nb_cc == 1 ) return( 1 ); else return( 0 );
}
```

Figure 4: The initial, small but not efficient, function used for code generation

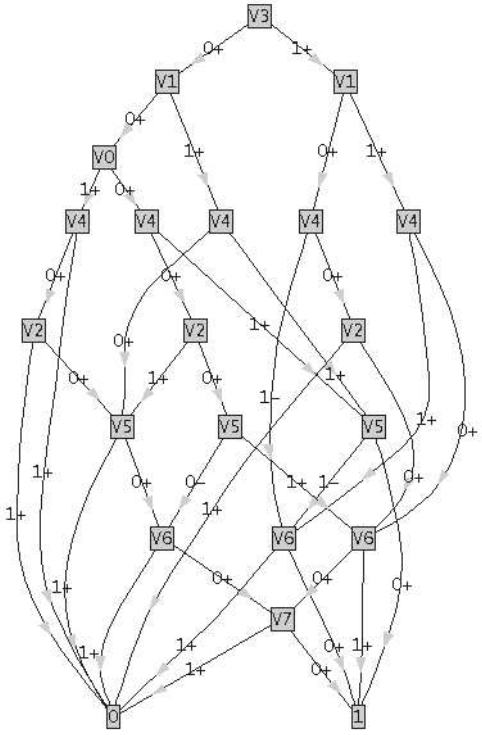

Figure 5: The BDD produced after optimization of the number of nodes through variable reordering. The THEN (resp. ELSE) edges are labeled by 1 (resp. 0). The sign indicates the type of the edge (cf section 2).

```
int is_simple_2D_BDD_1 (const int *V) {
int R = 0;
  L993: if (VAL(V,3)) {goto L929;} else {goto L656;}
  L929: if (VAL(V,1)) {goto L641;} else {goto L896;}
  L641: if (VAL(V,4)) {goto L305;} else {goto L528;}
  L305: if (VAL(V,6)) {goto L081;} else {goto L080;}
  L528: if (VAL(V,6)) {goto L080;} else {goto L337;}
  L337: if (VAL(V,7)) {goto L081;} else {goto L080;}
  L896: if (VAL(V,4)) {R=(!R);goto L305;} else {goto L625;}
  L625: if (VAL(V,2)) {goto L081;} else {goto L528;}
  L656: if (VAL(V,1)) {goto L832;} else {goto L705;}
  L832: if (VAL(V,4)) {goto L864;} else {goto L545;}
  L864: if (VAL(V,5)) {R=(!R);goto L305;} else {goto L080;}
  L545: if (VAL(V,5)) {goto L081;} else {goto L849;}
  L849: if (VAL(V,6)) {goto L081;} else {goto L337;}
  L705: if (VAL(V,0)) {goto L961;} else {goto L352;}
  L961: if (VAL(V,4)) {goto L081;} else {goto L609;}
  L609: if (VAL(V,2)) {goto L081;} else {goto L545;}
  L352: if (VAL(V,4)) {goto L864;} else {goto L769;}
  L769: if (VAL(V,2)) {goto L545;} else {goto L560;}
  L560: if (VAL(V,5)) {goto L528;} else {R=(!R);goto L849;}
  L080: return !R;
  L081: return R;
}
```

Figure 6: Function generated using the first scheme (see text).

to 503 nodes. Using the second strategy presented in section 3, we generated a 1500-line piece of C-code. We then compared the generated code to the initial one in terms of computational time, by evaluating the function over all the potential variable (2^{26}) assignements.

A first comparison between the straightforward implementation of the boolean characterization and the generated code is given in table 2. Histograms of the numbers of tests are given in figure 8. Table 3 shows the average time required for one function evaluation on several architectures.

	minimum number of tests	average number of tests	maximum number of tests
bool. charac. of [3]	6	111.41	176
bool. func. (BDDs)	6	8.71	26

Table 2: Comparison of the generated code with the boolean characterization of [3] for the 3D simplicity test over the 2^{26} potential neighborhoods.

```
int is_simple_2D_BDD_2 (const int *V) {
  int R = 0;
  if ( V[3] )
    if ( V[1] )
      if ( V[4] )
        return ( V[6] ? R : !R ) ;
      else
        Lbl0: if ( V[6] )
          return ( !R ) ;
        else
          return ( V[7] ? R : !R ) ;
    else
      if ( V[4])
        return ( V[6] ? !R : R ) ;
      else
        if ( V[2] )
          return ( R ) ;
        else
          { goto Lbl0; }
  else
    if ( V[1] )
      if ( V[4] )
        Lbl1: if ( V[5] )
          return ( V[6] ? !R : R ) ;
        else
          return ( !R ) ;
      else
        Lbl2: if ( V[5] )
          return ( R ) ;
        else
          Lbl3: if ( V[6] )
            return ( R ) ;
          else
            return ( V[7] ? R : !R ) ;
    else
      if ( V[0] )
        if ( V[4] )
          return ( R ) ;
        else
          if ( V[2] )
            return ( R ) ;
          else
            {goto Lbl2;}
      else
        if ( V[4] )
          {goto Lbl1;}
        else
          if ( V[2] )
            {goto Lbl2;}
          else
            if ( V[5] )
              {goto Lbl0;}
            else
              {R= !R;goto Lbl3;}
}
```

Figure 7: Function generated using the second scheme (see text).

Machine characteristics	Time (μsec)
SUN Ultra1 Sparc, 167MHz, SUNOS5.5	0.211
SGI Indy IP22, 150MHz, IRIX5.3	0.45
DEC Alpha 3000, 166MHz, OSF1V3.2	0.276

Table 3: Average time for one function evaluation (see text).

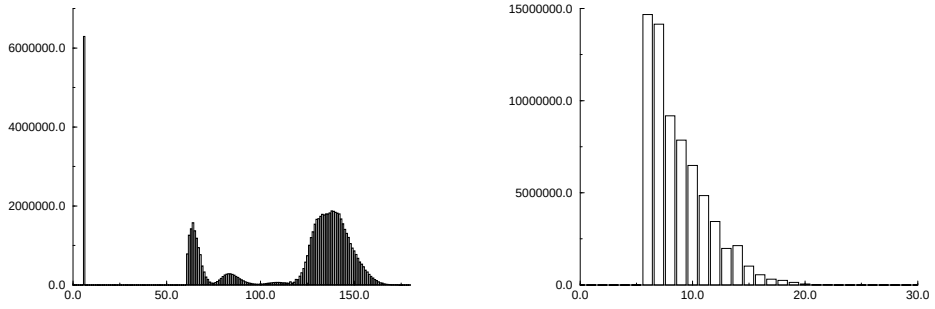

Figure 8: Histograms of the numbers of tests needed to evaluate the 3D simplicity test over the 2^{26} possible neighborhoods. Left: boolean characterization [3]. Right: code produced using BDD's.

4.5 Thinning in 3D: first method

Thinning a binary image consists of deleting simple points which are not end points. In general, from the implementation standpoint, the two tests are performed sequentially. For instance, the deletion condition described in [2] relies on the boolean characterization described in [3] and an extra condition which tells if the simple point is an end point. This new constraint increases the number of tests (compare tables 2 and 4, and left histograms of figure 8 and figure 9).

It turns out that the corresponding BDD is much smaller than the one computed in the previous section (only 272 nodes), which in turn decreases the number of tests performed by the generated code. This last result is illustrated by table 4: the minimum, average and maximum numbers of tests to decide whether a point can be deleted are significantly smaller than for simple point detection.

	minimum number of tests	average number of tests	maximum number of tests
deletion cond. of [2]	30	114.10	176
bool. func. (BDDs)	2	5.16	25

Table 4: Comparison of the generated code with the deletion conditions [2] over the 2^{26} possible neighborhoods.

4.6 Thinning in 3D: second method

For efficiency, some 3D thinning algorithms do not use directly the 3D simplicity test, but only sufficient – and computationally simpler – conditions. After these sufficient conditions have been checked, an additional condition prevents end points from being deleted.

The parallel thinning algorithm described in [11] is one of these. At each iteration, the algorithm updates all the voxel values in parallel. Each iteration is divided into six stages. Each stage detects border points along one of the 6 directions (N, S, E, W, Up, Down) and checks if these points can be deleted. Five conditions are proposed in [11] for performing the test. Since these conditions are easily expressed in terms of boolean expressions, we generated the BDD using the method described in

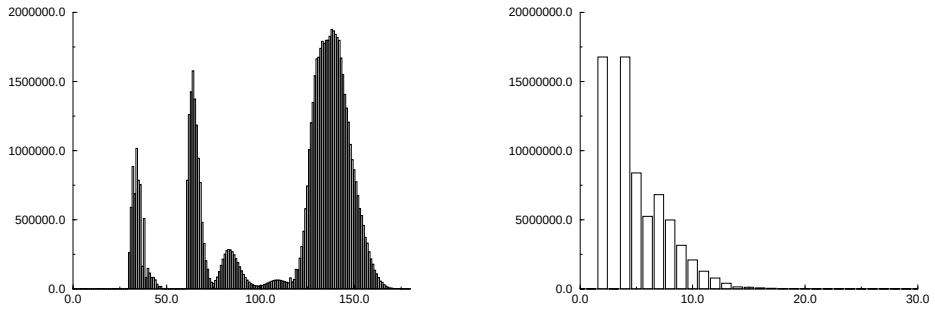

Figure 9: Histograms of the numbers of tests involved in checking the deletion condition over the 2^{26} possible neighborhoods. Left: boolean characterization [2]. Right: code produced using BDD's.

section 3.1.1. We compare here the direct, straightforward implementation of these 5 conditions and the corresponding code generated using BDD's. The results are presented in table 5 and figure 10.

	minimum number of tests	average number of tests	maximum number of tests
deletion cond. of [11]	1	4.83	56
bool. func. (BDDs)	1	3.27	26

Table 5: Comparison of the generated code with the deletion conditions [11] over the 2^{26} possible configurations.

5 Conclusion and future work

This article describes an approach for automatically generating efficient code for region-based binary image processing algorithms. Inspired from techniques used in digital system design, this approach can be applied to any binary image processing algorithm which evaluates a discrete function over small regions of the image

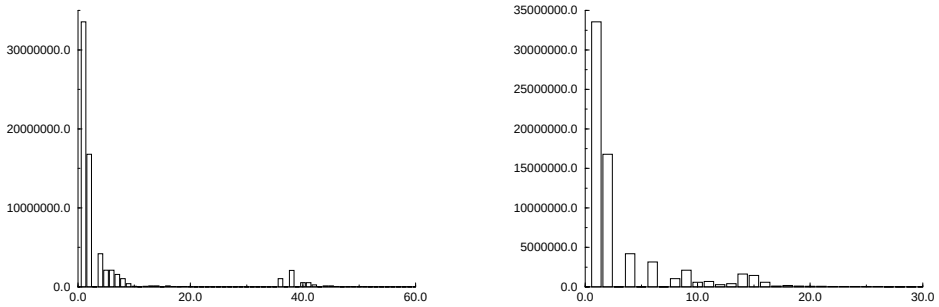

Figure 10: Histograms of the number of tests needed to determine if a point can be deleted [11] over the 2^{26} possible configurations. Left: direct implementation. Right: code produced using BDD's.

(e.g., discrete morphological and topological operations). Given a description of the function in terms of either boolean formulae or a compiled module, it automatically produces a program which implements the function. The generated C source code is portable and compact. It is also very efficient: at each stage of its execution, the procedure is guaranteed to examine only the pertinent input data, i.e., the values which affect the result. For each such value, it performs at most one test, one branching and one binary register operation.

We applied our method to several binary image processing tasks, such as the 2D and 3D simplicity tests, and two different 3D thinning processes. In each case, we produced functions more efficient than the previously optimized implementations, reducing the execution time by a factor of up to 20.

There remain a number of directions still to be explored. First, there exist many other image processing applications in which the technique described in this paper can be directly applied (edges tracking, classification of corners and junctions, etc). Second, more efficiency can be reached by studying more carefully the distribution of the incoming data (depending on the type of images or or the algorithm) and taking it into consideration when optimizing the BDD. Third, a better knowledge of the various efficient data structures used in digital design would probably help us address other image processing tasks, or deal with other kinds of images (grey-level, floating-point).

We are convinced that there remains a lot to gain in bringing the power of structures such as BDD's to the domains of computer vision and image analysis.

Acknowledgements

We would like to thank Ellen Sentovitch for all the precious informations about BDD's. Also, she carefully read previous versions of the paper and gave us lots of valuable comments. Thanks to Gérard Berry for helpful discussions.

References

[1] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C(27):509–516, June 1978.

[2] G. Bertrand. A parallel thinning algorithm for medial surfaces. *Pattern Recognition Letters*, 16:979–986, 1995.

[3] G. Bertrand. Boolean characterization of 3D simple points. *Pattern Recognition Letters*, 17:115–124, 1996.

[4] G. Bertrand and G. Malandain. A new characterization of three-dimensional simple points. *Pattern Recognition Letters*, 15(2):169–175, February 1994.

[5] J.-P. Billon. Perfect normal forms for discrete functions. Research Report 87019, BULL, March 1987.

[6] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a bdd package. In *Proc. of the 27th IEEE Design Automation Conference*, pages 40–45, Orlando, Fl., June 1990.

[7] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–317, September 1992.

[8] C. Castelluccia and W. Dabbous. Hippco: A high performance protocol code optimizer. Research Report 2478, INRIA, December 1995.

[9] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, A. Sangiovanni-Vincentelli, and E. Sentovich. Synthesis of software programs for embedded control applications. In *Proc. 32^{nd} Design Automation Conference*, June 1995.

[10] Gérard Giraudon. A efficient edge following algorithm. In *5th Scandinavian Conference on Image Analysis, Stockholm*, June 1987.

[11] W.X. Gong and G. Bertrand. A simple parallel 3-D thinning algorithm. In *10th International Conference on Pattern Recognition*, Atlantic City, June 17–21 1990.

[12] R.W. Hall and C.-Y. Hu. Time-efficient compuattion of 3d topological functions. *Pattern Recognition Letters*, 17:1017–1033, 1996.

[13] T.Y. Kong and A. Rosenfeld. Digital topology: introduction and survey. *Computer Vision, Graphics, and Image Processing*, 48:357–393, 1989.

[14] L. Lam, S.-W. Lee, and C. Y. Suen. Thinning Methodologies – A Comprehensive Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(9):869–885, september 1992.

[15] L. Latecki and C.C. Ma. An algorithm for a 3d simplicity test. *Computer Vision and Image Understanding*, 63(2):388–393, March 1996.

[16] C.Y. Lee. Representation of switching functions by binary decision programs. *Bell Systems Technical Journal*, (38):985–999, 1959.

[17] T.-C. Lee, R.L. Kashyap, and C.-N. Chu. Building skeleton models via 3-d medial surface/axis thinning algorithms. *Computer Vision and Image Understanding*, 56(6):462–478, November 1994.

[18] G. Malandain and G. Bertrand. Fast characterization of 3-D simple points. In *11th International Conference on Pattern Recognition*, The Hague, The Netherlands, August 30 – September 3 1992. IAPR.

[19] G. Malandain, G. Bertrand, and N. Ayache. Topological segmentation of discrete surfaces. *International Journal of Computer Vision*, 10(2):183–197, 1993.

[20] D.G. Morgenthaler. Three-dimensional simple points: serial erosion, parallel thinning, and skeletonization. Tr–1005, Computer Science Center, University of Maryland, College Park, MD 20742, U.S.A., February 1981.

[21] A. Nakamura and K. Aizawa. On the recognition of properties of three-dimensional pictures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:708–713, November 1985.

[22] M. Otte and H.-H Nagel. Extraction of line drawings from gray value images by non-local analysis of edge element structures. In G. Sandini, editor, *Proceedings of the 2nd European Conference on Computer Vision*, volume 588 of *Lecture Notes in Computer Science*, pages 687–695, Santa Margherita Ligure, Italy, May 1992. Springer-Verlag.

[23] A. Rosenfeld. Three-dimensional digital topology. Tr–936, Computer Science Center, University of Maryland, College Park, MD 20742, U.S.A., September 1980.

[24] P.K. Saha and B.B. Chaudhuri. 3d digital topology under binary transformation with applications. *Computer Vision and Image Understanding*, 63(3):418–429, 1996.

[25] P.K. Saha, B.B. Chaudhuri, B. Chanda, and D.D. Majumder. Topology preservation in 3D digital space. *Pattern Recognition*, 27(2):295–300, 1994.

[26] J. Serra. *Image analysis and mathematical morphology*, volume 1. Academic Press, 1982.

[27] J. Serra. *Image analysis and mathematical morphology: theoretical advances*, volume 2. Academic Press, 1988.

[28] J.I. Toriwaki, S. Yokoi, T. Yonekura, and T. Fukumura. Topological properties and topology-preserving transformation of a three-dimensional binary picture. In *6th International Conference on Pattern Recognition*, pages 414–419, Munich, October 1982.

[29] Y.F. Tsao and K.S. Fu. A 3-D parallel skeletonization thinning algorithm. *IEEE PRIP Conference*, pages 678–683, 1982.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399