Algorithms for Generalized Digital Images Represented by Bintrees

Hanspeter Bieri
Igor Metz

IAM-91-001

March 1991

Algorithms for Generalized Digital Images Represented by Bintrees

Hanspeter Bieri* Igor Metz*

Abstract

Generalized digital images, subsequently called hyperimages, represent a variation of the conventional digital images which implies pixels of different dimensions within the same image. The extent of a hyperimage is the disjoint union of all pixel extents it contains, which are relatively open unit cubes with respect to the euclidean topology of the underlying space. This approach is independent of any specific dimension of image and space, respectively, and allows strict partitioning of images into subimages, not just subdividing.

Since the storage required by a d-dimensional hyperimage of resolution n^d is $\approx 2^d n^d$ when using a binary matrix representation, a more space efficient bintree representation is investigated. Algorithms for the Boolean operations, the computation of elementary topological properties and the computation of some important measures of d-dimensional hyperimages (volume, surface, Euler characteristic) are presented. Because of the nature of bintrees, the implementation of these algorithms, too, can be performed independently of any specific dimension of image and space.

1 Binary Digital Images and Hyperimages

A d-dimensional *binary digital image* can most easily be modelled by a d-dimensional binary array. Each element of the array represents a (d-dimensional) *pixel* which is normally called "black" ("white") if the element's value is 1 (0). For most purposes it is necessary to introduce topological notions (adjacency, connectedness, etc.) for digital images and to study their properties. This is the main topic of *digital topology* [KR89]. Depending on the specific point of view, pixels are then normally understood as elements of $\mathbb{Z}^d$, as points with integer coordinates in $\mathbb{R}^d$, or as d-dimensional closed unit cubes. The resulting difficulties are well-known and rather easily explained (cf. [Pav82], [BN84], and especially [Kov89]).

[Bie90] avoids these difficulties by giving up the requirement that all pixels of a digital image have to be of the same type. As this approach does not conform to the conventional definitions of digital images [Fiu89], the resulting "images" are called *generalized binary digital images* or *hyperimages*. [Bie90] starts from the pixel understood as a d-dimensional closed unit cube and replaces it by pixels which are *relatively open unit cubes of dimensions* $0, 1, ..., d$. For the case $d = 2$, Figure 1 shows that an "old" pixel is the *disjoint union* of 9 "new" pixels of which four are 0-dimensional, four 1-dimensional, and one 2-dimensional. It is practical to distinguish between the "horizontal" and "vertical" 1-dimensional "new" pixels. Consequently we get for $d = 2$ four types of pixels which we denote by the symbols ∘, —, |, □. Figure 2 shows a conventional 2-dimensional binary digital image, with its 36 pixels understood as closed unit squares. Figure 3 shows the corresponding hyperimage where the numbers of pixels belonging to the four types are 49, 42, 42, and 36. The two figures clearly show the most important advantage of hyperimages compared to conventional digital

*Institut für Informatik und angewandte Mathematik, Universität Bern, Länggassstrasse 51, 3012 Bern, Switzerland. Email: bieri@iam.unibe.ch, metz@iam.unibe.ch

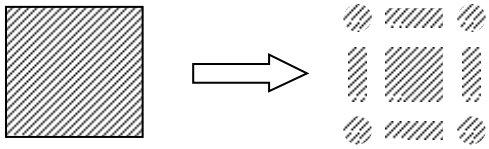

Figure 1: An "old" pixel partitioned by 9 "new" pixels.

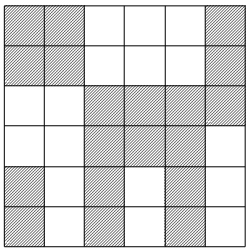

Figure 2: A conventional digital image.

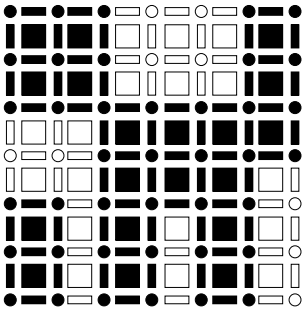

Figure 3: A hyperimage (not normalized).

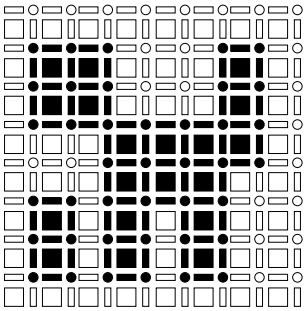

Figure 4: A hyperimage to be represented by a bintree.

images: The "old" pixels in Figure 2 form only a *subdivision* of the whole image (different "old" pixels are not necessarily disjoint sets), whereas the "new" pixels in Figure 3 form a *partition* of the hyperimage.

Like a conventional digital image, also a hyperimage can most easily be modelled by a d-dimensional binary array. Again, every element of the array represents a pixel which is understood as a unit cube $\subset \mathbb{R}^d$. But now, all these unit cubes are relatively open and their dimension is no longer necessarily $= d$. In order to get a normalized representation, we assume without loss of generality that the first element in the array (i. e. the element with all its indices $= 1$) represents a d-dimensional pixel and that the number of elements along each axis is odd. With the first assumption, an element represents a d-dimensional pixel iff all its indices are $= 1 \bmod 2$. All other elements represent (relatively open) *faces* of such d-dimensional unit cubes, with dimensions $\in \{0, \ldots, d-1\}$. They are uniquely determined by the requirement that all pixels together must form a *partition* of the extent of the whole hyperimage which is a d-dimensional box $\subset \mathbb{R}^d$. For a normalized hyperimage this box is an open set.

[Bie90] shows that the euclidean topology of $\mathbb{R}^d$ induces on every d-dimensional hyperimage a finite topology which can easily be described, e. g. by characterizing the smallest open neighborhood of every pixel type. This topology allows the design of elegant algorithms. It also represents a natural illustration of the main proposition in [Kov89]. A more complicated but equivalent approach has been proposed in [Kha70, KKM].

While the concept of hyperimages is not restricted to any specific dimension, the following sections will only deal with algorithms for hyperimages of dimension $d = 2$. This restriction is insignificant and its only purpose is ease of demonstration. The nature of the underlying data structures, i.e. arrays and bintrees, allows likewise an immediate generalization of these algorithms to higher dimensions.

2 Matrix and Bintree Representations

Representing a hyperimage by a binary array is straightforward as we have seen above. Given a $m \times n$ binary array `Mat1` representing a conventional binary image, we generate from it a $(2m+3) \times (2n+3)$ binary array `Mat2` which represents the corresponding normalized hyperimage. The conversion procedure works as follows:

- Allocate a binary array `Mat2[1..2m+3, 1..2n+3]`.
- For each element `Mat1[i,j]` which is set to `true`, set to `true` all elements `Mat2[i',j']` with $i' \in \{2i, 2i+1, 2i+2\}$, $j' \in \{2j, 2j+1, 2j+2\}$. Set all other elements to `false`.

(Please note that in all figures the first matrix element, i. e. the one having both indices = 1, is shown in the lower left corner.)

To determine the type of a pixel in the resulting array the indices can be used:

i'	j'	type of pixel
odd	odd	□
odd	even	|
even	odd	—
even	even	∘

After that, to reduce the amount of space used to store hyperimages we are going to use a *hierarchical data structure technique* to represent this kind of spatial data. We prefer *bintrees* [ST85] to *quadtrees* [Sam90] because they allow an elegant implementation of algorithms which perform independently of any specific dimension of image or space. We further decide to use *pointer bintrees* instead of *linear bintrees* mainly because it is easier to implement pointer based algorithms. Pointer based algorithms have been viewed as being storage wasters, but fairly compact pointer based implementations are possible [SW89].

A straightforward implementation of a bintree node in C++ [Lip89] might look as follows[1]:

```
enum tNodeType {GRAY, WHITE, BLACK};

struct tNode {
  tNode *left, *right;
  tNodeType NodeType;
  tNode(tNodeType nt, tNode* l, tNode* r)
            { NodeType = nt; left = l; right = r; }
};
```

To convert a binary array `Mat2` representing a hyperimage which corresponds to a $m \times n$ binary digital image into a pointer based bintree, we proceed as follows:

- Allocate a $2^k \times 2^k$ binary array `Mat3` where $k = \lceil \log(\max(2m+3, 2n+3)) \rceil$.
- Embed `Mat2` in `Mat3` such that `Mat3[i,j] = Mat2[i,j]` for $i = 1, \ldots, 2m+3$, $j = 1, \ldots, 2n+3$ and `Mat3[i,j] = false` for all other values of i and j.
- We can now consider this new binary array as an ordinary $2^k \times 2^k$ image array and convert it into a pointer bintree using any known method [Sam90].

The enlarged hyperimage represented by the array `Mat3` which is used to construct the bintree for our previous example is shown in Figure 4.

[1] The record `tNode` defines a so-called *constructor*, a function having the same name as the record. The constructor is used in conjunction with the `new` operator for memory allocation. It is called automatically to initialize the record slots after the memory for the record has been allocated.

3 Boolean Operations

Boolean or set-theoretic operations are often used as basic functions for modelling tasks in image processing or computer graphics. Since all Boolean operations on bintree representations of hyperimages are based on the same idea, we present only one of them: the intersection operator. Boolean operations on hyperimages do not need to consider the pixel types, their bintree implementations are identical, therefore, to those corresponding to conventional binary images.

The algorithm traverses the two input bintrees in parallel and examines corresponding nodes yielding the output bintree. An implementation in C++ is shown below. The example in Figure 5 shows the intersection of the two digitized letters A and B.

```
tNode*
Intersect(tNode *tree1, tNode *tree2)
{
  if ( (tree1→NodeType == tree2→NodeType) && (tree1→NodeType == GRAY)) {
    // Both subtrees are gray. Compute the intersection of their corresponding
    // subtrees and put the pointers to the results into variables 'left' and 'right'.

    tNode *left = Intersect(tree1→left, tree2→left);
    tNode *right = Intersect(tree1→right, tree2→right);

    if ((left→NodeType == right→NodeType) && (left→NodeType ≠ GRAY)) {
      // Both nodes are leaves, and they have the same color. Delete
      // one of them and return the other,
      delete right; // 'right' not needed any more
      return left;
    }
    else // Return a tree which has 'left' and 'right' as its subtrees.
      return new tNode(GRAY, left, right);
  }

  // One of the subtrees is a leaf and is white.
  if ((tree1→NodeType == WHITE) || (tree2→NodeType == WHITE))
      return new tNode(WHITE, NULL, NULL);

  if (tree1→NodeType == BLACK)
    return copy(tree2); // Return a copy of tree2
  else
    return copy(tree1); // Return a copy of tree1

}

tNode*
copy(tNode *t)
     // Returns a copy of the tree rooted at 't'.
{
  if (t == NULL)
    return NULL;
  else
    return new tNode(t→NodeType, copy(t→left), copy(t→right));
}
```

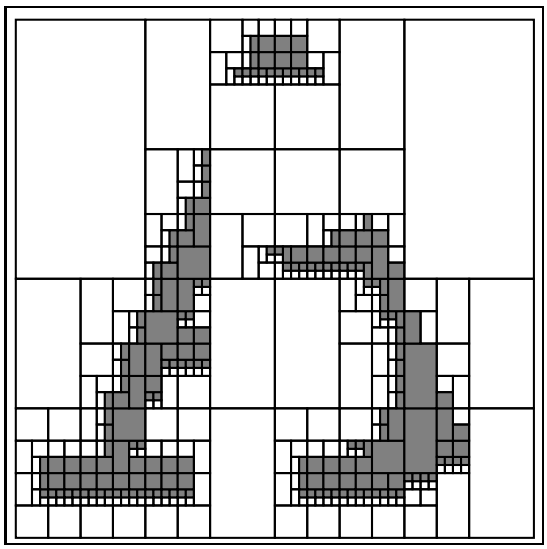

Figure 5: The intersection of the two digitized letters A and B.

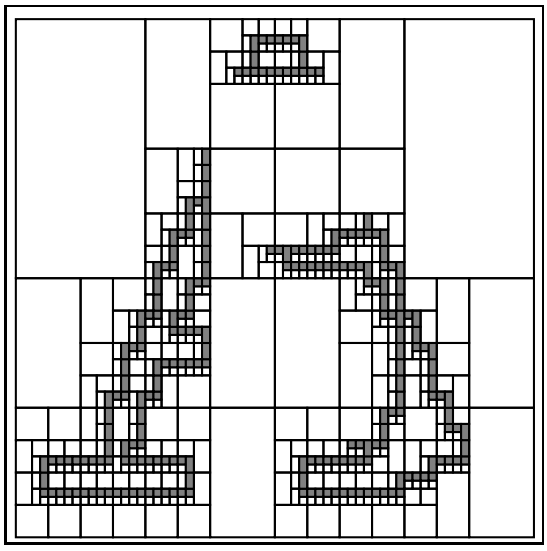

Figure 6: The boundary of the hyperimage in Figure 5.

4 Topological Operations

We will describe algorithms for the construction of the *boundary*, the *interior* and the *closure* of a hyperimage which are again hyperimages (cf. [Bie90]). The algorithms, as presented in this paper, operate in situ, i. e. they modify the input tree.

All three algorithms consist of two steps which both include a bintree traversal in preorder. The first step partitions the given hyperimage into extents which wholly belong to the *exterior*, to the *boundary* or to the *interior*, respectively, of the black part of the image. The second step is a postprocessing step and converts the intermediate result of the first step into a bintree representing the boundary, the interior or the closure, respectively, of the given hyperimage.

4.1 First step: Partitioning the image

Every leaf in the bintree represents an extent of the given hyperimage. The goal of this first step is to determine if a node represents an extent which wholly belongs to the *exterior*, to the *boundary* or to the *interior*, respectively.

During this step we traverse the tree in preorder. Figure 7 deals with the black leaves and Figure 8 with the white leaves we may meet. The extent represented by a leaf (first column) and its surrounding pattern (second column) are examined, and an appropriate action (third or fourth column, respectively) is taken. Either the node can be marked by a `B` (boundary), an `I` (interior) or an `E` (exterior), or it has to be refined by an additional bintree. This latter case is indicated by a bold arrow.

We will not dicuss all cases shown in Figures 7 and 8 in detail. Discussing two of them should be enough to make the procedure evident.

1. Let the leaf reached represent a black extent of type `o` (first line in Figure 7). If all its neighbours are black, then this leaf is marked as belonging to the interior of the hyperimage. Otherwise it will be marked as belonging to the boundary.
2. Let the leaf reached represent a quadratic black extent of length > 1 (last two lines in Figure 7).

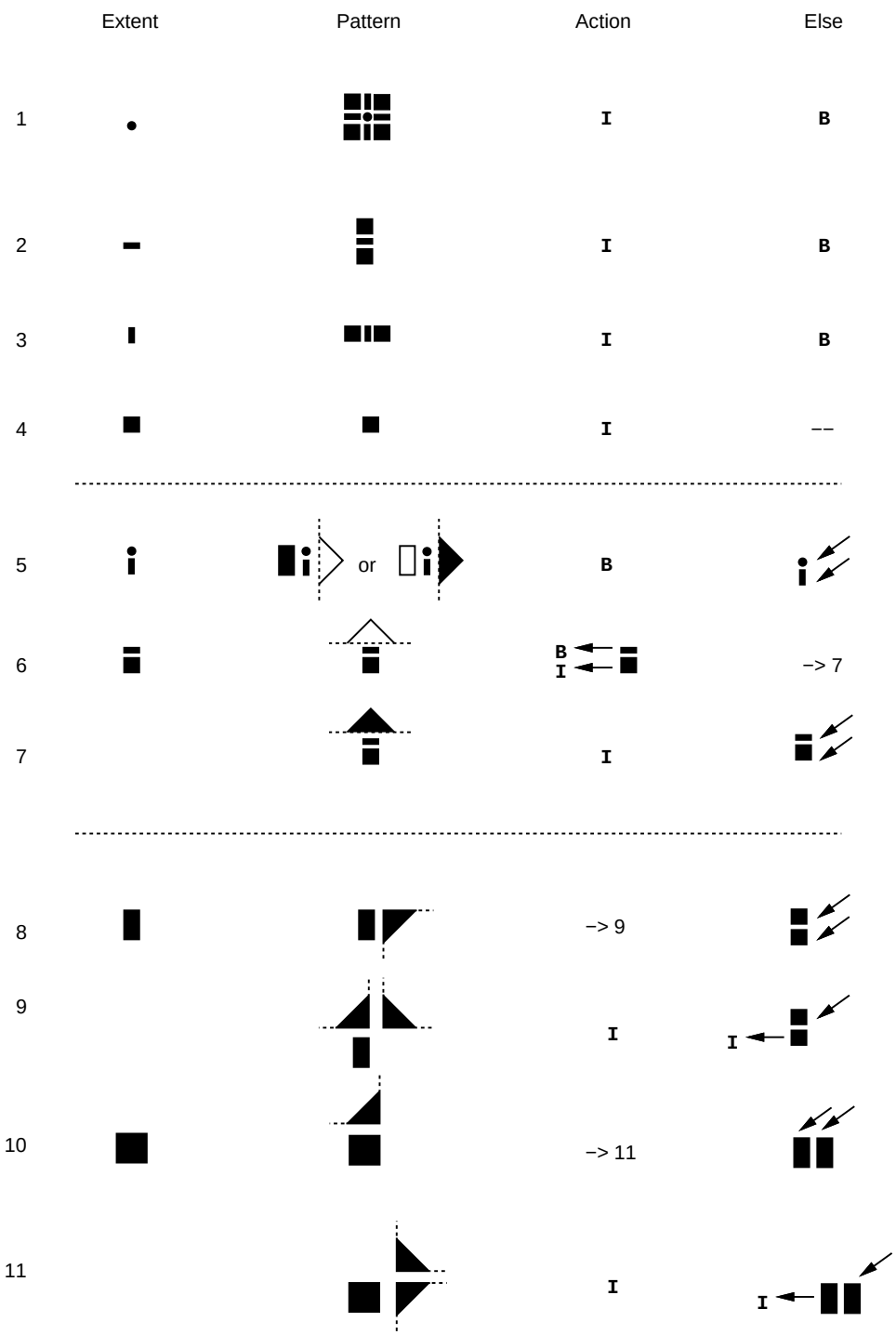

Figure 7: Partitioning of black extents.

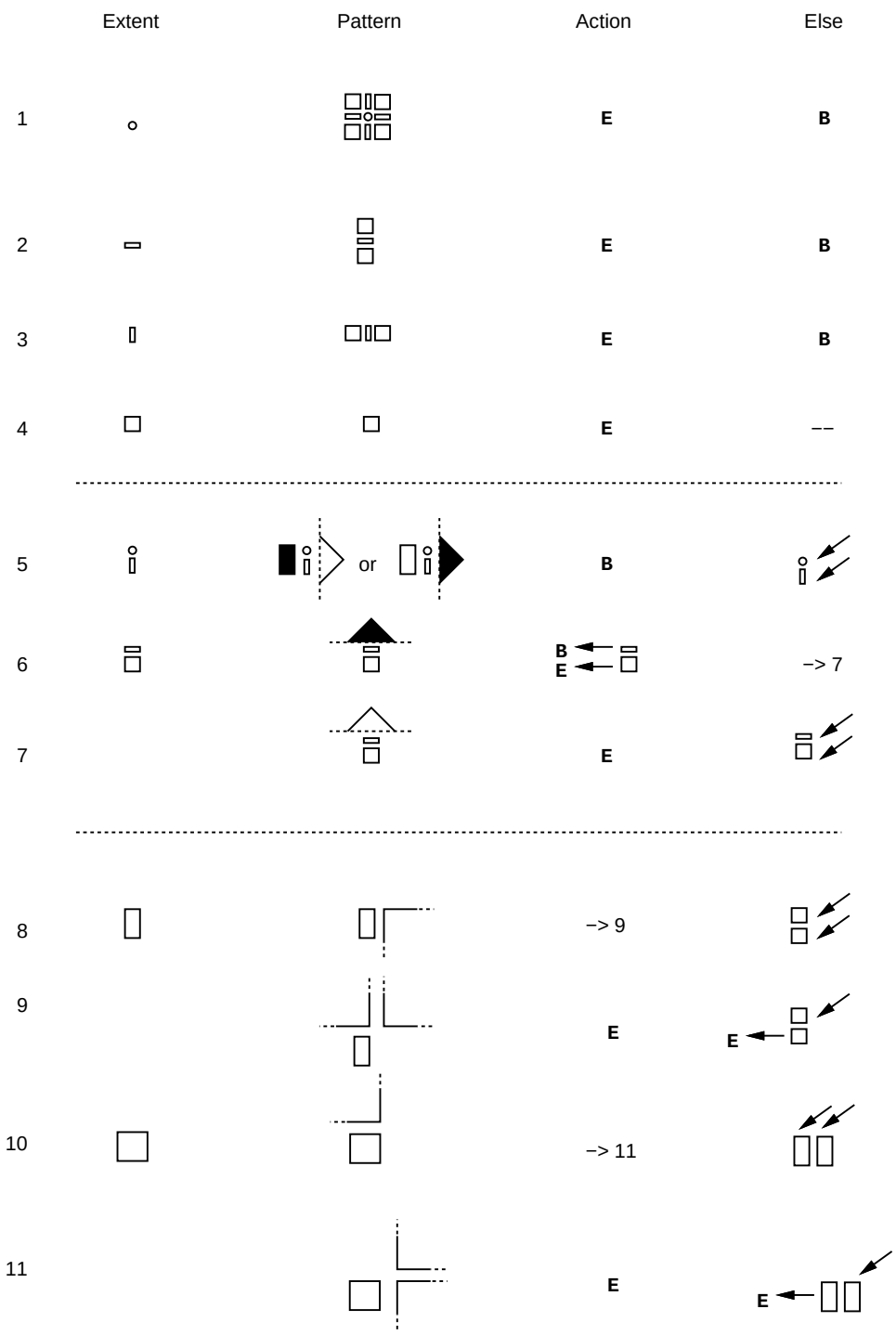

Figure 8: Partitioning of white extents.

If the upper extent is not black we refine this node, i. e. we replace it by a bintree consisting of three nodes, and recursively check the left and the right leaf.
Else if the upper right extent and the right extent are both black, the node is marked as belonging to the interior of the hyperimage.
Otherwise we refine this node, i. e. we replace it by a bintree consisting of three nodes, and recursively check the right leaf. The left leaf will be marked as belonging to the interior of the hyperimage.

4.2 Computing the Boundary

To compute the boundary, we must first partition both the white and the black extents of a hyperimage (= first step, consisting of one traversal). In the postprocessing step we traverse the intermediate tree and colour all nodes black which have been marked by a `B`. All other nodes will be coloured white. Neighbouring regions of the same colour will be merged. The result is a hyperimage which is the boundary of the input hyperimage. An example is shown in Figure 6.

This procedure can be simplified for hyperimages, which have been generated from conventional binary images. In this case it is sufficient to consider only the black extents, as such hyperimages are always closed.

4.3 Computing the Interior

To compute the interior of a hyperimage, we only need to partition its black extents. In the postprocessing step we traverse the intermediate tree and change the colour of all black nodes which have been marked by a `B` to white, and merge neighbouring regions of the same colour. The result is a hyperimage which is the interior of the input hyperimage.

4.4 Computing the Closure

To compute the closure of a hyperimage, we only need to partition its white extents. In the postprocessing step we traverse the intermediate tree and change the colour of all white nodes which have been marked by a `B` to black, and merge neighbouring regions of the same colour. The result is a hyperimage which is the closure of the input hyperimage.

5 Algorithm for some Geometric Properties of Hyperimages

The area, the surface and the Euler number of digital images can be computed by means of the so-called *quermassintegrals* [BN84]. Hyperimages are more suited to apply this technique than conventional digital images, for quermassintegrals are *additive functionals* and to compute them recursively is easier for images which can be partitioned into subimages (i. e. different subimages are really disjoint) than for images which can only be subdivided.

Let E be any pixel of dimension $d \in \{0,1,2\}$ in a 2-dimensional hyperimage. The values

$$W_r(E) = (-1)^{r+d} \frac{\binom{r+d}{2}}{\binom{r+d}{d}} \omega_r ,$$

where $r = 0,1,2$ and $\omega_0 = 1, \omega_1 = 2, \omega_2 = \pi$ are the *quermassintegrals* of E.

Let $n_{\circ}, n_{—}, n_{|}$, and $n_{\square}$ be the number of pixels of type $\circ$, —, |, and $\square$, respectively. As quermassintegrals are additive, the quermassintegral W_0, W_1, W_2 of a hyperimage can be computed simply by the following equation:

$$W_i = W_i(\circ) n_{\circ} + W_i(—) n_{—} + W_i(|) n_{|} + W_i(\square) n_{\square} \qquad (i = 0,1,2)$$

Quermassintegrals are useful in image analysis mainly because they are related to the area A, the circumference C and the Euler number χ of an image in the following way:

$$\begin{aligned} A(R) &= W_0(R), \\ C(R) &= 2\,W_1(R), \\ \chi(R) &= \frac{1}{\pi} W_2(R), \end{aligned}$$

where R denotes the *figure* or *black region* of the respective (hyper)image. Computing the above values for the black region of the hyperimage in Figure 5 results in an area of 112, a circumference of 160, and an Euler number of 3.

$W_r(\circ), W_r(—), W_r(|)$ and $W_r(\square)$ must only be computed once. The values of $n_\circ, n—, n_|$ and $n\square$ which are necessary, too, to compute the quermassintegrals can be determined by a simple procedure which traverses the bintree in preorder.

```
PROC Count_Pixel_Types(tree : POINTER TO tNode; VAR n∘,n—, n|, n□: INTEGER);
BEGIN
    IF tree→NodeType = GRAY THEN
        Count_Pixel_Types(tree→left, n∘, n—, n|, n□);
        Count_Pixel_Types(tree→left, n∘, n—, n|, n□);
    ELSIF tree→NodeType = BLACK THEN
        IF "maximum depth of tree is reached" THEN
            CASE Type_Of(tree) OF
            ∘:   n∘ := n∘ + 1;
            —:   n— := n— + 1;
            |:   n| := n| + 1;
            □:   n□ := n□ + 1;
            END CASE;
        ELSIF "depth of maximum−1 is reached" THEN
            CASE Type_Of(tree) OF
            ∘
            |:   n∘ := n∘ + 1;
                 n| := n| + 1;
            ‾
            □:   n— := n— + 1;
                 n□ := n□ + 1;
            END CASE;
        ELSE
            "From current depth and resolution compute width and height of binbox
            represented by this node";
            FOR i ∈ {∘,—,|,□} DO
                ni := ni + width/2 height/2;
            END FOR;
        END IF;
    END IF;
END PROC;
```

6 Short Experimental Analysis of Space and Time Complexity

To compare the space requirements for implementing conventional images with those for hyperimages, we have performed some experiments. The input for the experiments consists of a set of 23 digitized uppercase letters with resolution 32×32 and a set of 79 cross-sections of

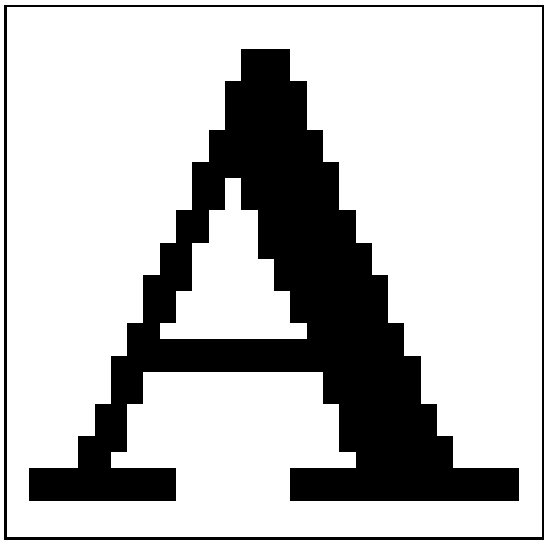

Figure 9: A sample from the dataset "Alphabet": the capital letter A.

Figure 10: A sample from the dataset "CT data": cross-section no. 43.

a human pelvis generated from computed tomographic (CT) data with resolution 128×128. Figures 9 and 10 show samples from both datasets.

While the binary array representation of a 2-dimensional hyperimage always consumes 4 times more space than the representation of the corresponding conventional image, the bintree representation needs "only" ≈ 2.35 times more space on the average for the dataset mentioned above. This aspect is of some interest for the respresentation of large images. The plots in figures 11 and 12 show the number of internal nodes used to store the conventional images and the hyperimages of our dataset.

Bintrees representing single cross-sections from the CT dataset have some 6000 nodes on the average. This gives an average space requirement of 54000 bytes using the node structure described earlier with two pointer fields and one nodetype field[2]. To store the same hyperimage in a bit array, we will need $(2 \times 128 + 3)^2 = 67081$ bits, or 8386 bytes. To store the hyperimage bitwise is very space efficient, but will yield considerable execution-time penalties. For this reason one would probably implement the hyperimage using a byte array, which will yield a space requirement of 67081 bytes. In order to perform the topological operations, space requirements will even double as we need a second array to store the result of the operation.

It is evident, that the execution time of all algorithms presented in this paper is linear with respect to the number of nodes of the input tree, as each node in the input tree is only visited once or twice. Execution time measurements have confirmed this statement. Figures 13 and 14 show scatterplots of the execution times measured on a Sun SPARCstation 1 using the GNU C++ Compiler version 1.37.2.

Operations on hyperimages can be much faster with array representations than with bintree representations. Using the digitized alphabet as input, our experiment showed that the array representation resulted in a roughly 10 times faster execution (on the average) than the bintree representation. Using the CT data as input, the factor was about 4. Since we didn't dispose of images with a higher resolution, we duplicated the pixels in the CT dataset

[2]This calculation is quite naive, as it ignores the fact that the memory allocation scheme of the underlying operating system requires some administrative overhead per allocated chunk, plus an alignment adjustment. This overhead can be reduced using a memory allocation scheme which is adapted to this specific application area.

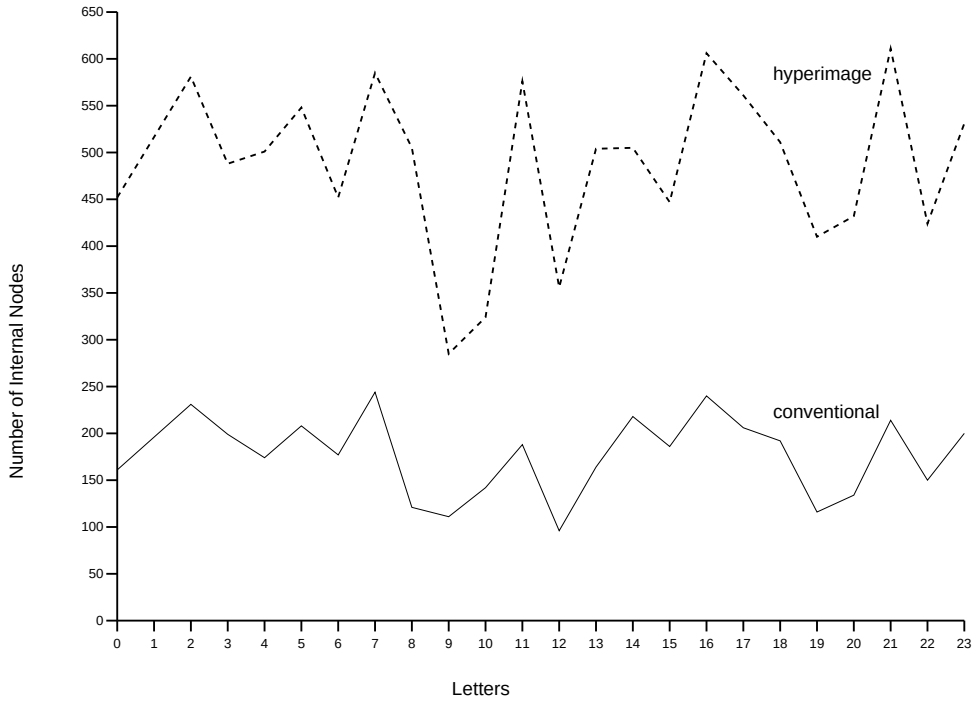

Figure 11: Space requirements for dataset "Alphabet"

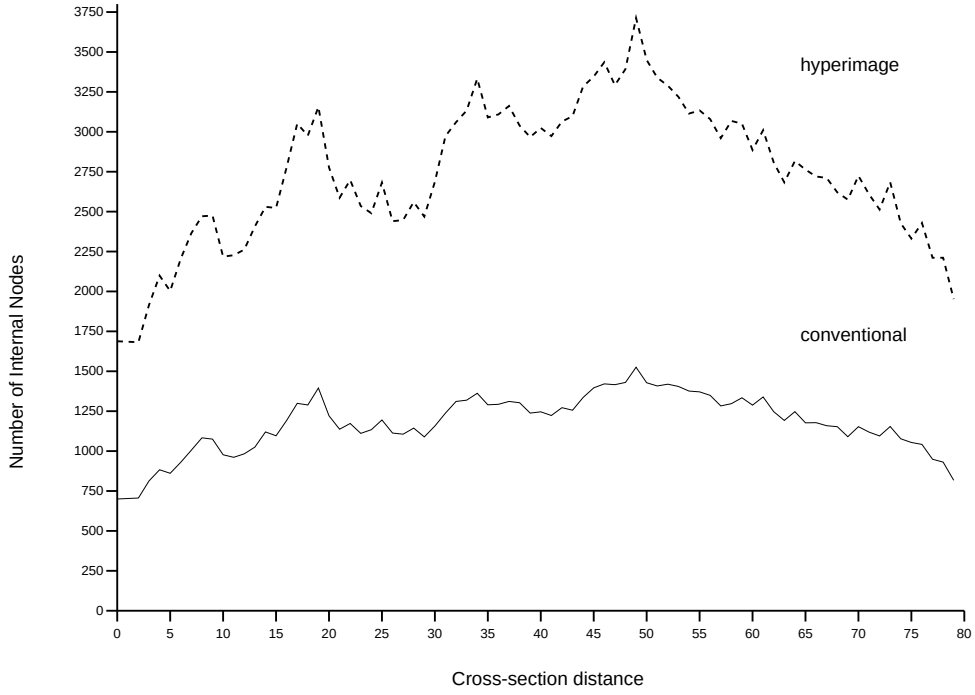

Figure 12: Space requirements for dataset "CT data"

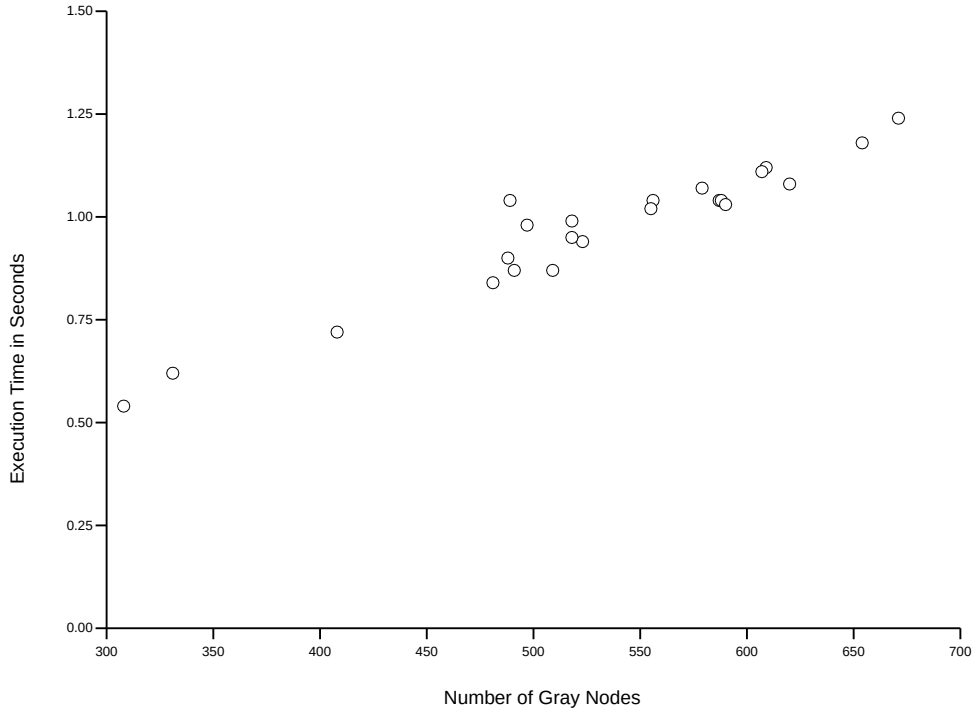

Figure 13: Execution times needed for computing the boundary of hyperimages from dataset "Alphabet".

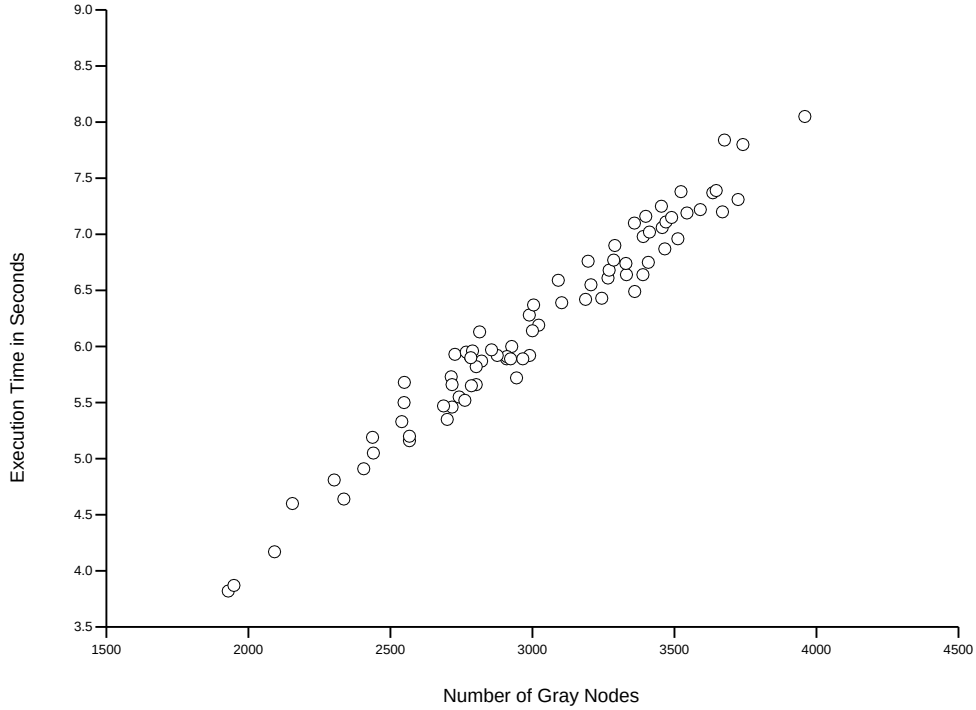

Figure 14: Execution times needed for computing the boundary of hyperimages from dataset "CT data".

in order to blow-up the images to a 256×256 resolution. For these blown-up images the execution times were almost equal for the two representations.

7 Conclusions

We have shown that the bintree representation of hyperimages reduces the space requirements and that it allows to implement efficiently many algorithms which are important for hyperimages. One other positive aspect of using bintrees is that all our algorithms can easily be generalized to higher dimensions. The application of the methods to three- or four-dimensional images should prove to be of practical interest.

Acknowledgement

We are grateful to Åke Wallin of the Institute for Biomechanics, University of Berne for providing us the CT data used for our experiments.

References

[Bie90] H. Bieri. Hyperimages - an alternative to the conventional digital images. In C.E. Vandoni and D.A. Duce, editors, *Proceedings of EUROGRAPHICS '90*, pages 341–352. North-Holland, 1990.

[BN84] H. Bieri and W. Nef. Algorithms for the Euler characteristic and related additive functionals of digital objects. *Computer Vision, Graphics, and Image Processing*, 28:166–175, 1984.

[Fiu89] E.L. Fiume. *The Mathematical Structure of Raster Graphics.* Academic Press, 1989.

[Kha70] E. D. Khalimsky. Applications of connected ordered topological spaces in topology. *Conference of math. departments of Povolsia*, 1970.

[KKM] T. Y. Kong, R. Kopperman, and P. R. Meyer. A topological approach to digital topology. *To appear*

[KR89] T.Y. Kong and A. Rosenfeld. Digital topology: Introduction and survey. *Computer Vision, Graphics, and Image Processing*, 48:357–393, 1989.

[Kov89] V.A. Kovalevsky. Finite topology as applied to image analysis. *Computer Vision, Graphics, and Image Processing*, 46:141–161, 1989.

[Lip89] S. B. Lippman. *C++ Primer.* Addison-Wesley, 1989.

[Pav82] T. Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press, 1982.

[Sam90] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1990.

[ST85] H. Samet and M. Tamminen. Bintrees, CSG trees, and time. *Computer Graphics*, 19(3):121–130, 1985.

[SW89] H. Samet and R. E. Webber. A comparison of the space requirements of multi-dimensional quadtree-based file structures. *Visual Computer*, 5:349–359, 1989.