Compact Topological Map
Guillaume Damiand, Brandel Sylvain, J. Rossignac

To cite this version:
Guillaume Damiand, Brandel Sylvain, J. Rossignac. Compact Topological Map. [Research Report] LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon. 2017. hal-02017158

HAL Id: hal-02017158
https://hal.archives-ouvertes.fr/hal-02017158
Submitted on 13 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Compact Topological Map

Guillaume Damiand¹, Sylvain Brandel¹, and Jarek Rossignac²

¹ Univ Lyon, CNRS, LIRIS, UMR5205, F-69622 France
² School of Interactive Computing, Georgia Institute of Technology, Atlanta, USA

Abstract. Several work have shown that a topological map is a good tool to describe and handle a labeled image. Indeed, it allows describing the full topology of the image: its subdivision in cells (vertices, edges, faces) and all the incidence and adjacency relations between these cells, contrary to simpler models like region adjacency graphs. Thanks to this description, a topological map allows computing and updating several features that can mix topological and geometrical information. One major problem of topological map, that often limits its use, is the memory size required by its representation. In this paper, we solve this drawback by proposing two compact representations of topological maps that offer different space/time efficiency compromises. These two representations are based on the links between topological maps and linel maps, where each edge corresponds to a linel, and on an implicit encoding of linel darts. Our experiments show a major reduction on memory space comparing to previous encoding, and allows us to envisage to process big images without memory constraint.

Keywords: 2D Image Representation; Multi-Label Image; Combinatorial Maps; Topological Maps.

1 Introduction

In order to describe the content of a 2D image, for example to analyze its content or to track moving objects, image’s pixels are often grouped in homogeneous area, for a given criterion, for example by segmenting the image. This grouping of pixels can be described by a labeled image, an image where each pixel is associated with a label (which can be a number, a name, a physical property. . . ). Then tools must be defined in order to represent a labeled image, to compute features on the proposed representation and operations must be designed in order to handle and update the representation. Moreover, memory and time complexity of data-structures and algorithms must be, as usual, the best possible.

Several work have studied the question of data-structures and operations to represent a labeled image. One first solution was the region adjacency graph (RAG) [18] where each region (a maximal connected set of pixels having the same label) of a labeled image is described by a vertex in the graph, and an edge exists between two vertices if the corresponding two regions are adjacent. Thanks to this graph, it is possible to test if two regions \( R_1 \) and \( R_2 \) are adjacent in linear time regarding the number of regions adjacent to \( R_1 \) (or \( R_2 \)). Moreover,
when two adjacent regions are merged, the graph can be updated accordingly only by contracting the corresponding edge. Despite these advantages, a region adjacency graph has several drawbacks: it does not describe multi-adjacency relations (i.e. if two regions are adjacent several times), and it does not describe the order of the adjacency relations (it is not possible to iterate through all the regions adjacent to a given region in an ordered way).

**Dual-graphs** [14] solve these limitations. This is a pair of multi-graph. A first one, $G$, is the extension of a RAG to represent multiple adjacency relations. The second one is the dual of $G$ (the graph having one vertex per face of $G$, and one edge between two vertices when the two corresponding faces of $G$ are adjacent). Another solution is the ordered graph [12] that describes a graph plus the order of edges around each vertex. Several other solutions have been proposed based on combinatorial maps [17]. A 2D combinatorial map can be seen as an oriented graph that represents also the order of edges around vertices and thus can be used to describe and handle a labeled image [10, 3].

The main advantages of combinatorial maps are to be defined in any dimension and to have consistency constraints that guarantee the topological validity of represented objects. Moreover many operations exist that allow to compute features and to modify described objects [8]. For all of these reasons, many work have used combinatorial maps in 2D and 3D image processing algorithms, defining topological maps for the specialization of combinatorial maps to represent discrete images. Topological maps were used for example for image segmentation [4, 2], polygonal approximation [7], objects detection [1] or to compute topological invariants [9], sometimes by using multi-levels approaches [5, 11].

All these advantages and these work show that a topological map is a good tool to handle a labeled image. However, one main drawback of topological maps, that often limits its use, is the memory size required by its representation which can be too big to deal with large images. The main contribution of this work solves this drawback by proposing two compact representations of topological maps that offer different space/time efficiency compromises. These two solutions give two choices to users to favor memory or speed depending on their needs.

In this paper, we start to introduce preliminary notions in Sect. 2. In Sect. 3, we define lineal combinatorial map, called $l$-map, and show that it is possible to retrieve all the information of a topological map thanks to its corresponding $l$-map. This allows us to define an implicit representation of $l$-maps in Sect. 4 by using only pointels and local numbering. This implicit representation is then used in Sect. 5 in order to define our two compact encodings of $l$-maps, which give directly, thanks to the link with topological maps, a compact encoding of topological maps. We present experiments in Sect. 6 in order to evaluate our solutions and conclude and give some future works in Sect. 7.

### 2 Preliminary Notions

A pixel is an element of $\mathbb{N}^2$. Two pixels $p = (x, y)$ and $p' = (x', y')$ are **4-adjacent** if $|x - x'| + |y - y'| = 1$. A **4-path** between two pixels $p$ and $p'$ is a
A sequence of pixels \((p = p_1, \ldots, p_k = p')\) such that each pair of consecutive pixels are 4-adjacent. A set of pixels \(S\) is **4-connected** if there is a 4-path between any pair of pixels in \(S\) having all its pixels in \(S\).

An **image** \(I\) is a finite set of pixels (the image domain), and a mapping between \(I\) and a set of colors or gray levels (the pixel values). In a **labeled image**, each pixel \(p \in I\) is also associated with a label \(l(p)\), a value in a given finite set \(L\). A **region** in a labeled image is a maximal 4-connected set of pixels having the same label. Regions form a partition of the image: two different regions are disjointed, and the union of all the regions is the whole image. A specific region, called **infinite region**, and denoted \(R_0\), is the complement of \(I\). A region \(R_1\) is **enclosed** in a region \(R_2\) if each 4-path between any pixel of \(R_1\) and any pixel of \(R_0\) has at least one pixel in \(R_2\) (intuitively a path starting from \(R_1\) and going to the exterior necessarily goes through \(R_2\)). \(R_1\) is **directly enclosed** in \(R_2\) if there is no region \(R_3 \neq R_2\) with \(R_1\) enclosed in \(R_3\) and \(R_3\) enclosed in \(R_2\).

**Interpixel** [13] topology considers the cellular decomposition of the euclidean space \(\mathbb{R}^2\): pixels are elements of dimension 2 (unit squares) of the decomposition, **linels** are elements of dimension 1 (unit segments) between pixels, and **pointels** are elements of dimension 0 (points) between linels.

In 2D, a **combinatorial map**, called **2-map**, describes a subdivision of the plane (or of a surface) in cells (vertices, edges and faces), and describes all the incidence and adjacency relations between these cells. All the cells are described by using a single basic element, called **dart** (drawn by oriented curves in Fig. 1(b) and numbered). Two mappings are defined on these darts: \(\beta_1(d)\) gives, for each dart, the next dart in the same face than \(d\); \(\beta_2(d)\) gives, for each dart, the other dart in the same edge but in the other face than \(d\) (for example, in Fig. 1(b), \(\beta_1(5) = 6\) and \(\beta_2(5) = 8\)). We denote by \(\beta_0 = \beta_1^{-1}\) (note this is not a third mapping but only a notation): allowing to retrieve the previous dart of a given dart in the same face (for example, in Fig. 1(b), \(\beta_0(5) = 4\)).

A 2-map can be used to describe a labeled image \(I\) through the notion of **topological map**: a triplet \(TM = (C, T, P)\) (see [6] for exact definitions and an example in Fig. 1). (1) \(C\) is a 2-map such that each face of \(C\) corresponds to a boundary of a region of \(I\). Furthermore \(C\) is minimal in its number of darts. Thanks to this minimality, each edge of \(C\) corresponds to a maximal frontier between two adjacent regions. (2) \(T\) is an enclosure tree of regions: each node of
Fig. 2. (a) The linel combinatorial map of the topological map shown in Fig. 1. (b) An example of edge split to illustrate the links between l-darts and darts.

Let us consider $D_0 = \{d \in D_l \mid \text{pointel}(d) \text{ is on in } P \}$ (the subset of l-darts having their pointel on, l-darts with red dots in Fig. 2). There is a one to one mapping between l-darts of $D_0$ and darts of $C$ (the minimal combinatorial map of $TM$). Indeed, all darts in $C$ are associated with pointels on; and insertion
operation does not switch any new pointel on. In the following of this paper, we use this one to one mapping to simplify our notations, denoting \( d \) both for an original dart in \( C \) and for its corresponding \( l \)-dart in \( C^l \).

The following relationships between \( \beta_i \) mappings and \( \beta_i^l \) mappings can be given knowing that each edge in \( C \) is split in several unit edges in \( C^l \), and that all new \( l \)-darts have their pointel off: \( \forall d \in D_o \):

- \( \beta_1(d) = \beta_1^{l^k} \circ \beta_1^l(d), \) \( k \geq 0 \) is the smallest integer such that \( \beta_1^{l^k} \circ \beta_1^l(d) \in D_o \);
- \( \beta_2(d) = \beta_0^{l^k} \circ \beta_2^l(d), \) \( k \geq 0 \) is the smallest integer such that \( \beta_0^{l^k} \circ \beta_2^l(d) \in D_o \).

\( \beta_1^{l^k} \) means apply \( \beta_1 \) \( k \) times. Since \( \beta_0 = \beta_1^{-1} \), we have \( \beta_0(d) = \beta_0^l(d) \circ \beta_1^{l^k} \).

These relationships are illustrated in the example given in Fig. 2(b). The initial edge made of the two darts \( \{1, 2\} \) is split in eighteen \( l \)-darts (black oriented segments in the figure). \( \beta_1(1) = 3 \), the \( l \)-dart obtained from \( \beta_1^l(1) \) and using \( \beta_1^l \) eight times. When following this path of \( l \)-darts, all pointels are off (because the corresponding \( l \)-darts result of the split of the initial edge) except the last pointel of \( l \)-dart 3. Things are similar for \( \beta_0(4) = 2 \), the \( l \)-dart obtained starting from \( \beta_0^l(4) \) and using \( \beta_0^l \) eight times; and for \( \beta_2(1) = 2 \), the \( l \)-dart obtained from \( \beta_2^l(1) \) and using \( \beta_0^l \) eight times.

These relationships give us directly the three algorithms that take a dart \( d \) as parameter and compute \( \beta_i(d) \) by using only \( l \)-darts, \( \beta_i^l \) mappings, and the test if a pointel is off. Algorithm 1 allows to compute \( \beta_1(d) \). Note that this algorithm takes a dart \( d \) as input, and thus pointel\( (d) \) is on by definition. Algorithms to compute \( \beta_0(d) \) and \( \beta_2(d) \) are similar.

One major advantage of a linel combinatorial map, and its relationships with a corresponding topological map, is the possibility to be considered at two different layers. (1) We can use \( \beta_i^l \) links and iterate through a 1-map exactly as if it is were a topological map. This is useful to compute easily “topological information” as the number of times two regions are adjacent or to retrieve all the regions adjacent to a given one. (2) We can use the \( \beta_i^l \) links that allow to iterate through the “geometry” of the topological map; for example to draw the contours of a region by iterating through all its linels.

### 4 Implicit Representation of Linel Combinatorial Map

A \( l \)-dart \( d \) can be uniquely labeled by a pair \( (p, n) \), called descriptor, and denoted by \( \text{desc}(d) \). \( p \) is the end-pointel of \( d \), and \( n \) is a number between 0 and 3: 0 for the bottom \( l \)-dart around \( p \), 1 for the right \( l \)-dart, 2 for the top \( l \)-dart and 3 for the left \( l \)-dart (cf. Fig. 3(a)). Given \( \text{desc}(d) \), it is straightforward to retrieve pointel\( (d) \) and linel\( (d) \), which are directly encoded in the descriptor.

Now we define the three algorithms that compute the \( \beta_i^l \) mappings by using only descriptors and test if a linel is on. Algorithm 2 allows to compute \( \beta_2^l(d) \).
Fig. 3. (a) Descriptors for \(l\)-darts. Pointel \((0, 0)\), in black, can be the pointel of four \(l\)-darts (black oriented segments), numbered locally from 0 to 3. The two \(l\)-darts to the right of this pointel have descriptors \(((0, 0), 1)\) for the black \(l\)-dart and \(((1, 0), 3)\) for the green \(l\)-dart. (b) Example to illustrate \(\beta_l(d)\) computation. Pointels are drawn by dots, red for pointels \(on\), and blue for pointels \(off\).

Algorithm 2: Compute \(\beta_l^2(d)\).
\[
\text{Input: } ((i, j), n): \text{descriptor of } d. \\
\text{Output: } \text{descriptor of } \beta_l^2(d). \\
\text{switch } n \text{ do} \\
\text{case 0: } j \leftarrow j + 1; \ n \leftarrow 2; \\
\text{case 1: } i \leftarrow i + 1; \ n \leftarrow 3; \\
\text{case 2: } j \leftarrow j - 1; \ n \leftarrow 0; \\
\text{case 3: } i \leftarrow i - 1; \ n \leftarrow 1; \\
\text{return } ((i, j), n)
\]

This algorithm tests the four possible cases of \(l\)-darts, by using the local \(l\)-dart number, and returns directly the descriptor of \(\beta_l^2(d)\) depending on each case. For example, if \(n = 0\), \(\beta_l^2(d)\) is the dart with local number 2 and having pointel \((i, j + 1)\) as end-pointel.

In the example given in Fig. 3(b), \(\beta_l^2(1) = 14\) with \(\text{desc}(1) = ((1, 1), 3)\) and \(\text{desc}(14) = ((0, 1), 1)\) (case 3 in Algo. 2). Another example is \(\beta_l^2(2) = 13\) with \(\text{desc}(2) = ((1, 0), 0)\) and \(\text{desc}(13) = ((1, 1), 2)\) (case 0 in Algo. 2).

Algorithm 3 allows to compute \(\beta_l^1(d)\). We turn around \(p\), the end-pointel of \(d\), in counter-clockwise order, until finding a linel \(l'\), incident to \(p\) (same principle is used in digital boundary tracking, for example in [15]). We turn counter-clockwise because external contours are oriented clockwise, thus each \(l\)-dart has its region to its right (this is also true for inner contours because they are oriented counter-clockwise). \(\beta_l^1(d)\) is the \(l\)-dart having \(l'\) as linel and pointel \((d)\) as end-pointel: this is the \(l\)-dart \(\beta_l^2(p, n)\). Note that by definition it is not possible to have only one linel \(on\) around a pointel. Thus starting from a \(l\)-dart (having hence its linel \(on\) \(l\)), we necessarily find a second linel \(on\) \(l' \neq l\).

If we look at Fig. 3(a), we can observe that given a black \(l\)-dart \(d\) as input, \(\beta_l^1(d)\) is necessarily a colored \(l\)-dart because \(d\) and its \(\beta_l^1(d)\) have necessarily two different end-pointels. For example \(\beta_l^1(3)\) is necessarily one \(l\)-dart among \(((0, 1), 2), ((1, 0), 3), ((0, -1), 0)\) and \(((1, 0), 1)\), depending on the local configuration of linels \(on\) around the black pointel.
In the example given in Fig. 3(b), $\beta_1(1) = 2$ with $\text{desc}(1) = ((1,1),3)$ and $\text{desc}(2) = ((1,0),0)$. In the iteration around pointel $(1,1)$, the first linel on found, after linel(1), is linel(13) ($\text{desc}(13) = ((1,1),2)$); then $\beta_2(13) = 2$ is returned.

The algorithm to compute $\beta_0(d)$ is similar to the one that computes $\beta_1(d)$ by reversing the orders of operations: we start to use $\beta_2(p,n)$, then iterate clockwise around the end-pointel of $d$.

5 Compact Encoding of $l$-maps

In this section, we present two compact encodings of $l$-maps combinatorial maps. Thanks to the relationships between $l$-maps and topological maps, the two encoding of $l$-maps give us directly two compact encodings of topological maps. Note that in both cases, our encoding describes both the minimal combinatorial map $C$ and the interpixel matrix $P$ of the topological map within the same data-structure, increasing even more the memory space reduction.

5.1 Matrix Representation

In this version, a $l$-map is encoded by a matrix $M$ of three Booleans. $M[i,j]$ encodes the configuration around pointel $p = (i,j)$ by three Booleans: the first Boolean represents pointel $p$, the second Boolean the bottom linel of $p$ and the third Boolean the right pointel of $p$. The value of each Boolean is false if the corresponding element is off, and true if it is on.

The main advantage of this encoding is to provide a direct access to each pointel/linel. Testing if pointel $(i,j)$ is on is achieved directly by testing the first bit of $M[i][j]$. To test if linel $((i,j),n)$ is on, three cases must be considered. (1) If $n = 0$ or $n = 1$, testing if linel $((i,j),n)$ is on is directly the value of bit number $n+1$ of $M[i][j]$. (2) If $n = 2$, testing if linel $((i,j),2)$ is on is done by testing linel $((i,j-1),0)$. Indeed the top linel of pointel $(i,j)$ is the same than the bottom linel of pointel $(i,j-1)$; but is only represented in pointel $(i,j-1)$. (3) Similarly, if $n = 3$, testing if linel $((i,j),3)$ is on is done by testing linel $((i-1,j),1)$.

Another main advantage of this encoding is to allow simple modification operations. Indeed, it is enough to switch on/off some pointels/linels in order to modify the underlying topological map. Similarly, extraction algorithm becomes much simpler since it only consists to iterate through a labeled image and to switch on some pointels and linels based on local properties.

The main drawback of this representation is to “lose” some memory space to represent empty cells. The second representation proposed in this paper solves this drawback by do not representing at all the empty cells, but this is to the detriment of operations complexity.

5.2 Stacked Row Representation

The goal of this second version is to not store anything for empty cells. To do so, a $l$-map is now encoded by an array $R$ of bitset. $R[j]$ is the concatenation
of the encoding of all the active pointels of row number \( j \). A pointel is active either if it is on, or if it is incident to one bottom, right or top linel on. Each active pointel is encoded by four Booleans: the first Boolean represents pointel \( p \), the second Boolean the bottom linel of \( p \), the third Boolean the right linel of \( p \) and the fourth Boolean the top linel of \( p \). The value of each Boolean is false if the corresponding element is off, and true if it is on.

With this solution, each row is now encoded by a variable length representation which has \( 4n_j \) bits, \( n_j \) being the number of active pointels in row number \( j \). For this reason, it is no more possible to access directly, like for the previous solution, to a given pointel \( p = (i, j) \). We are going to explain now how to retrieve the neighbor pointels of \( p \) when we use this compact representation.

First, each \( l \)-dart descriptor \( ((i, j), n) \) is replaced by a descriptor, \( ((a, j), n) \), where \( a \) is the index of pointel \( p = (i, j) \) among the list of active pointels in the \( j \)th row (\( a = 0 \) for the first active pointel whatever its position, \( a = 1 \) for the second one, \ldots). This index allows to directly retrieve the four bits associated with pointel \( p \) which start at position \( 4a \) in \( R[j] \). This index allows also to retrieve directly the previous and the next pointel of \( p \) in row \( j \) which are pointels \( (a - 1, j) \) and \( (a + 1, j) \). Note that these two formula are only valid if pointel \( p \) is active and its previous or next pointel is also active, but these formula will only be used in these cases.

Things are more complex to find pointels above and below \( p \). Indeed, the number of active pointels in two consecutive rows are not necessarily the same, and thus it is not possible to use index \( a \) of row \( j \) to deduce index \( a' \) in row \( j - 1 \) or \( j + 1 \). One solution is to encode, after each active pointel, the index of the pointels above and below it; but in this case the memory reduction compared to the matrix based solution becomes very small and is thus not really interesting.

Our solution is to recompute these indices instead of storing them. Given an active pointel \( p = (a, j) \), the following principle is used to retrieve \( p' = (a', j - 1) \), the active pointel above \( p \). (1) We count the number \( n_1 \) of top linels on, before or equal to \( p \) in row \( j \). (2) We iterate through active pointels of row \( j - 1 \), starting from \( a' = 0 \), incrementing \( a' \), and counting the number \( n_2 \) of bottom linels on. When \( n_1 = n_2 \), the current pointel \( p' = (a', j - 1) \) is the active pointel above pointel \( p \). Same principle is applied to retrieve the pointel below \( p \) by counting bottom linels on in row \( j \), and top linels on in row \( j + 1 \).

Note that this encoding has some redundancy, because each vertical linel is stored twice: once as vertical linel above a given pointel \( (i, j) \); and once as vertical linel below pointel \( (i, j - 1) \). But this redundancy is necessary in order to retrieve up and down neighbors of a given pointel.

The main advantage of this solution is that no information is stored for non active pointels. Thus, the memory space is very compact and depends only on the size of the region boundaries, and not on the global size of the image. The main drawback is that the complexity to access to up and down neighbors pointel of a given pointel is no constant anymore but linear in number of active pointels.

Testing if pointel \( (a, j) \) is on is achieved directly by testing the bit at position \( 4a \) in \( R[j] \). Testing linels \( ((a, j), n) \) is also done directly when \( n = 0, 1 \) or 2: the
value of the linel is the bit at position $4a + n + 1$ in $R[j]$. For $n = 3$, testing if linel $((a, j), 3)$ is on is done by testing linel $((a - 1, j), 1)$.

5.3 Links Between Darts and Regions

The two previous sections have shown how to retrieve $\beta^l_i$, pointel and linel of any $l$-dart by using only dart descriptors and tests if pointels and linels are on or off. In order to represent all the information of a topological map, it misses us to retrieve region($d$) that gives the region that contains dart $d$.

To do so, an associative array, $A$, is added to our compact representation, that associates region identifiers to dart descriptors (e.g. a pointer or an index depending on how regions are stored). For each face of the $l$-map $C^l$, the minimal dart of the face is stored in $A$ (minimal by using a lexicographic order on descriptors). To retrieve the region of a dart $d$, given by its descriptor, we iterate through all the darts of its face, using $\beta_1$, until going back to $d$, and keeping the minimal dart, $d_{\min}$, encountered during this walk. The region of $d$ is directly obtained by $A[d_{\min}]$.

This solution has the main advantage to not require a big memory space since there is only one entry in $A$ for each face of $C^l$. Its main drawback is the complexity of the operation to retrieve region($d$) which is no more in constant time but linear in number of $l$-darts of the face containing $d$, plus the complexity to retrieve $d_{\min}$ in $A$. This complexity is linear in number of faces of $C^l$, $\#f$, when $A$ is represented by an array, is in $\log(\#f)$ when $A$ is sorted, and is in average constant time if $A$ is represented by a hash map.

6 Experiments

We have implemented our two compact representations of $l$-maps and compared memory spaces and computation times of our two solution (referenced in the following as compact storage with matrix or stacked row representation) with a classical implementation of 2D topological maps based on darts and pointers (referenced in the following as classical storage). We have used 224 images from [16] (arbogreens, cherries, football, greenlake and swissmountains classes) after a segmentation pre-processing. All these images have same resolution 756×504 (see two images in Fig. 4). We performed all benchmarks on the same computer, an Intel core-i7 4790 4-cores 3.6GHz with 32Go RAM.
We have first compared the memory space used by the three versions (see table 1). On average for all input images, the memory space used for regions tree is about the same for the three versions. In classical storage, each representative dart is stored as a pointer (8 bytes in 64bits architectures), while in compact storage, each dart is stored as a descriptor (2 integers and a character, that is 9 bytes), explaining the small difference between both implementations.

<table>
<thead>
<tr>
<th>Storage</th>
<th>classical</th>
<th>compact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>matrix representation</td>
<td>stacked row representation</td>
</tr>
<tr>
<td>Map storage</td>
<td>680,161</td>
<td>195,288</td>
</tr>
<tr>
<td>Region tree storage</td>
<td>192,409</td>
<td>202,536</td>
</tr>
</tbody>
</table>

Table 1. Mean of used memory space (Bytes) with 224 images.

The memory space used for the 2-map in compact storage with a matrix is about 3 times smaller than in classical storage, and compact storage with stacked row is about 2 times smaller than the one using a matrix. These differences are more important in particular cases. For example, if the number of regions is small compared to the image size, the map size using stacked row representation can be 20 times smaller than one using matrix representation. More precisely, since we store one pixel for each pixel in the matrix and we store only active pixels in the stacked row, the memory space reduction between both representations depends on the ratio between active pixels and all pixels. Similarly, the performances in terms of memory space of our three implementations strongly depend on the image contents: the memory space reduction of our compact representations is better for topological map having many small darts than for images having large regions represented by only few darts.

Second, we have compared computation times (see table 2). Accesses to all elements of a topological map are in constant time for classical storage. For matrix representation, \( \text{region}(d) \) is linear in number of dart per face, and the complexity of \( \beta_i \) mapping is linear in number of lines of the edge. For stacked row representation, the complexity of \( \beta_i \) mapping depends also on the number of active pixel of traversed rows. This explains why times are generally faster with classical storage than with our compact storages.

<table>
<thead>
<tr>
<th>Storage</th>
<th>classical</th>
<th>compact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>matrix representation</td>
<td>stacked row representation</td>
</tr>
<tr>
<td>Map extraction</td>
<td>91.6</td>
<td>10.2</td>
</tr>
<tr>
<td>Region tree building</td>
<td>0.5</td>
<td>38.9</td>
</tr>
<tr>
<td>All regions traversal</td>
<td>2.1</td>
<td>1.7</td>
</tr>
<tr>
<td>Segmentation</td>
<td>0.3</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table 2. Mean of computation time (ms) with 224 images.
We have analyzed the computation time for building topological maps from labeled images. This step is much faster with compact storage than with classical storage because compact storage computes only interpixel matrix, while classical storage computes additionally the 2-map. However, computation of the enclosure tree is faster with the classical representation due to the use of many $\beta_i$ operations.

Then, we have analyzed the computation time of two operations that iterate through a topological map $TM$. First we iterate through all regions of $TM$, and for each region $R$, we iterate through all its boundaries (as would do a method to visualize $TM$). In this case, the matrix representation is the fastest method because we directly iterate through $l$-darts, while with the classical representation we need to iterate through $TM$ then to each line of each dart (but the difference is very small). As planned, the stacked row representation is slower due to the additional time required by the $\beta_i$ operations.

The second operation analyzed consists in iterating through all regions of $TM$, and for each region $R$, through each dart in its boundaries. For each dart $d$, we test the color difference between region($d$) and region($\beta_2(d)$) and count as many differences are smaller than a given threshold (this is the first step of a region growing segmentation). Here, the method using the classical representation is about 10 times faster than method the method using the matrix representation, itself 10 times faster than the method using the stacked row representation.

*Our two compact representations give very good results in term or memory space occupation. Of course this is to the detriment of the complexity of operations.* But even with the stacked row representation, our results show that these complexities are still good enough to allow to process big images. And moreover, some operations are even faster than with the classical representation.

### 7 Conclusion

In this paper, we have proposed two compact representations of topological maps, solving the main drawback of these models which was the big memory space required to store them. These compact representations are based on the notion of linel combinatorial map and on relationships between darts and $\beta_i$ links of the topological map, and $l$-darts and $\beta_l^1$ links of the $l$-map. The first representation, based on an interpixel matrix, allows direct access to each interpixel element and simplifies the modification of the topological map. The second representation is very compact, avoiding to represent empty elements, but implies a bigger complexity of operations. Users can choose between these two representations depending on their priorities.

As future work, we plan to integrate modification algorithms in both proposed representations, like split and merge methods. It could be interesting to study the impact of the second representation on the computation time of these methods. Some algorithms should perhaps be modified in order to improve their complexities in this specific case. We plan to study other representations more compact than the matrix while trying to decrease the complexity of operations.
We also would like to decrease the memory space used by the region tree. Lastly, we want to extend these compact representations in 3D.

References